Common Services Framework: Users' Manual

Keith Cummings, John Hubbard, Steve Wampler, Bret Goodrich
Software Group

April 2014

	Name	*Signature*	*Date*
Approved By:	Bret Goodrich High Level Software Group Manager	*B. Goodrich*	*02-Apr-2014*
Approved By:	Rob Hubbard Systems Engineer	*R. Hubbard*	*03-Apr-2014*

REVISION SUMMARY:

1. Date: March 14, 2005
   Revision: A
   By: Bret Goodrich
   Changes: Initial PDR Version

2. Date: March 3, 2006
   Revision: A1
   By: Steve Wampler
   Changes: Panguitch-1 P2 version

3. Date: September 12, 2006
   Revision: A2
   By: Steve Wampler
   Changes: NSF PDR version

4. Date: January 8, 2008
   Revision: A3
   By: Steve Wampler
   Changes: Added to C++ API

5. Date: April 16, 2009
   Revision: A4
   By: Steve Wampler
   Changes: Brought up-to-date.

6. Date: October 28, 2009
   Revision: A5
   By: John Hubbard
   Changes: Added Property Tool section

7. Date: November 13, 2009
   Revision: A6
   By: John Hubbard
   Changes: Added Cache section

8. Date: December 24, 2009
   Revision: A7
   By: John Hubbard
   Changes: Fixed submit codes & Added code fragments to Controller example

9. Date: January 14, 2010
   Revision: A8
   By: Bret Goodrich
   Changes: Formatting

10. Date: January 26, 2010
    Revision: A9
    By: John Hubbard
    Changes: C++ details, removed property tools, minor cleanups and comments

11. Date: May 10, 2010
    Revision: A10
    By: John Hubbard
    Changes: Updated Alarm service helper methods

12. Date: July 12, 2010
    Revision: B
    By: John Hubbard
    Changes: Final Update prior to Canary 0 release (fixed c++ examples)

13. Date: September 3, 2010
    Revision: B
    By: John Hubbard
    Changes: Changes to the IFunctional interface

14. Date: December 22, 2010
    Revision: C
    By: John Hubbard
    Changes: Final Update prior to Canary 1 release

15. Date: April 2011
    Revision: D
    By: John Hubbard
    Changes: Canary 2 updates

16. Date: December 2011
    Revision: E
    By: John Hubbard
    Changes: Canary 3 updates

17. Date: June 2012
    Revision: F
    By: John Hubbard
    Changes: Canary 4 (new-health & others)

18. Date: Dec 2012
    Revision: G
    By: John Hubbard
    Changes: Canary 5 Updates & general clean up

19. Date: September 2013
    Revision: H
    By: Keith Cummings
    Changes: Canary 6 updates

20. Date: April 2014
    Revision: I
    By: Keith Cummings
    Changes: Canary 7 updates

TABLE OF CONTENTS

REVISION SUMMARY: 1
TABLE OF CONTENTS 3
1 Preface 7
2 Introduction 8
2.1 Background 8
2.2 Structure 8
2.3 Design Highlights 9
2.3.1 Communications-Neutral Architecture 9
2.3.2 Separation of Functional and Technical Behavior 9
2.3.3 Configuration-Driven Control 10
2.3.4 Container/Component Model 10
2.3.5 Service Toolboxes 11
3 Infrastructure 13
3.1 Communications 13
3.2 Attributes 13
3.2.1 Java Representation 14
3.2.2 C++ Representation 15
3.3 Attribute Tables 15
3.3.1 Java representation 16
3.3.2 C++ representation 17
3.4 Configurations 19
3.4.1 Java representation 19
3.4.2 C++ representation 20
3.5 Data Example 20
3.6 Java Example 20
3.7 C++ Example 21
3.8 Commands 21
3.9 Events 22
3.10 Containers and Components 23
3.10.1 Deploying a component 23
3.10.2 Managing a component's lifecycle 24
3.11 Services 24
4 Connection Service 26
4.1 Commands 26
4.2 Controller actions 27
4.3 Java helper 27
4.3.1 Java commands for IRemote 27
4.3.2 Java commands for IController 28
4.3.3 Java action callbacks 28
4.3.4 Java example 29
4.4 C++ helper 29
4.4.1 C++ commands for IRemote 30
4.4.2 C++ commands for IController 30
4.4.3 C++ commands for callbacks 30
4.4.4 C++ example 31
5 Event Service 32
5.1 Events 32

5.2 Event Strategies 32
5.2.1 Posting Strategies 32
5.2.2 Subscription Strategies 32
5.3 Event Posting 33
5.3.1 Java helper 33
5.3.2 C++ helper 34
5.4 Event Callbacks And Subscriptions 35
5.4.1 Java Helper 35
5.4.2 Java Example 35
5.4.3 C++ Helper 37
5.4.4 C++ Example 38
6 Log Service 40
6.1 Message Categories 40
6.2 Status Messages 41
6.3 Debug Messages 42
6.3.1 Debug Levels 42
6.4 Convenience Methods 42
6.4.1 Java Helper 42
6.4.2 Java Example 44
6.4.3 C++ Helper 45
6.4.4 C++ Example 46
7 Health Service 47
7.1 Java Helper 47
7.1.1 Java Example 48
7.2 C++ Helper 48
7.2.1 C++ Example 49
8 Property Service 50
8.1 Properties versus Constants 51
8.2 Component Access to Attribute Metadata 51
8.3 Java Property Service Helper 53
8.3.1 Java Example 56
8.4 C++ Helper 56
8.4.1 C++ Example 59
9 Alarm Service 60
9.1 Java Helper 60
9.2 C++ Helper 60
10 MINOR SERVICES 61
10.1 Archive Service 61
10.1.1 Java Helper 61
10.1.2 C++ Helper 61
10.2 Constant Service 62
10.2.1 Component Access to Manifest Constants 62
10.2.2 Java Property Service Helper 62
10.2.3 C++ Helper 63
10.3 User Interfaces Support 63
10.4 Miscellaneous Services 63
10.4.1 Thread Support 63
10.4.2 Generic Pools 63
10.4.3 ID Service 64
10.4.4 Cache 64
10.4.5 Time Service 65

10.4.6 Date Service 65
10.5 Scripting Support 65
10.5.1 Script Executors 66
10.5.2 Script Interpreters 66
10.5.3 Common Services Framework support for scripts 67
10.5.4 Java examples 68
10.6 Script Database 70
10.7 Parameter Set Database 71
11 Components 72
11.1 Component Lifecycles and Functionality 72
11.2 Component Lifecycle 72
11.2.1 Creation 72
11.2.2 Initialization 72
11.2.3 Startup 73
11.2.4 Operation 73
11.2.5 Shutdown 74
11.2.6 Un-initialization 74
11.2.7 Removal 74
11.3 Functional Architecture 74
11.4 Simulated Components 74
11.5 Java-based Components 75
11.5.1 Simulated Java Components 76
11.5.2 Java Example Component 76
11.6 C++-Based Components 76
11.6.1 Simulated C++ Components 77
11.6.2 C++ Example Component 78
12 Controllers 79
12.1 Controller Features 79
12.1.1 Command-Action-Response 79
12.1.2 Command Thread 80
12.1.3 Action Manager 81
12.1.4 Action Threads 82
12.1.5 Action Callbacks 82
12.1.6 Simulated Controllers 82
12.2 Control of Configuration Lifecycle 83
12.2.1 Scheduled 83
12.2.2 Running 83
12.2.3 Completed 83
12.3 Interface 84
12.3.1 Public Interface 85
12.3.2 Protected Interface 85
12.3.3 Attributes 86
12.3.4 Events 87
12.4 Action Callback Interface 87
12.4.1 Public Interface 88
12.4.2 Protected Interface 88
12.4.3 Controller Properties 88
12.5 Java-based Controllers 88
12.5.1 The Public Interface 88
12.5.2 The Protected Interface 89
12.5.3 Action Callbacks 90

12.6 Writing Java-based Controllers 90
12.6.1 The ControllerAdapter Class and Source File 91
12.6.2 Controller Example 91
12.6.3 The ActionCallbackAdapter Class and Source File 95
12.6.4 Action Callback Adapter Methods 95
12.7 C++-based Controllers 97
12.7.1 The Public Interface 97
12.7.2 The Protected Interface 97
12.7.3 Action Callbacks 98
12.8 Writing C++-based Controllers 99
12.8.1 The Controller Adapter Class and Source File 99
12.8.2 Controller Example 99
12.8.3 The Action Callback Adapter Class and Source File 104
12.8.4 Action Callback Adapter Methods 104

1 PREFACE

This document is the first of three parts that document the Common Services Framework (CSF). This section is a general user's manual and includes basic information about the use and extension of CSF. Part two is the reference guide, and includes more details about the technical bits of CSF the most developers probably do not need to know about. The third part is the API guide which is generated by the CSF make system.

This document should give a broad overview of CSF and its parts. If you are preparing to write code, consider looking at the Java doc or C++ Doxygen documentation as you go through this document. The Java doc and Doxygen are likely more verbose then this document when looking at specific methods and functions. See part 2 of this document (The Reference Guide) for details about how to build the Java doc and Doxygen.

Every attempt is made to keep this document correct and up to date, but sometimes it lags behind. If you discover errors, or omissions please contact the authors so that we can correct or include the missing information.

2 INTRODUCTION

2.1 BACKGROUND

Early in the conceptual design process ATST undertook a survey of observatory software control systems to determine the best approach to take on software design and implementation. A great deal of useful information was obtained as a result of this survey, one of which is that large, distributed, software projects can reduce their overall development, integration, and maintenance costs by basing as much software as possible on a standard infrastructure.

There are several viable models for this infrastructure in use in modern observatory systems. ATST has elected to use a *Common Services* model similar to that used for the Atacama Large Millimeter Array (ALMA) project Common Software (ACS). The ATST Common Services Framework (CSF) attempts to be more streamlined than the ACS and also less dependent on a specific *middleware* structure. This approach should allow the fundamental characteristics of CSF to be preserved as new middleware technologies are developed during the operating lifetime of ATST.

The benefits of a common services model for infrastructure include:

- All major system services are provided through standard interfaces used by all software packages. A small support team can thus support a number of development teams more easily.

- The separation of the functional and technical architectures provided by the common services model means that a significant amount of the technical architecture can be provided through the common services, allowing developers to concentrate on providing the functional behavior required of their software.

- There is a uniform implementation of the technical architecture across all systems. So long as the access to this technical architecture remains consistent, the implementation of the technical architecture can be modified with minimal impact on the development teams.

- Since application deployment is a technical issue and hence implemented within the common services, the software system as a whole is more easily managed within a distributed environment. This makes the use of less expensive, commodity computers more feasible.

2.2 STRUCTURE

The ATST Common Services Framework features are grouped into several broad categories:

- *Deployment support* – This support is implemented based on a Container/Component Model (CCM) and allows the uniform management of applications in a distributed system without regard to the functionality they provide. Base implementations for software components and controllers are provided as part of the deployment support. **All application functionality is implemented on top of these base implementations.**

- *Communications support* – These are services that are necessary or useful in a distributed system. They include:

    - Connection services that allow applications to communicate directly with other applications, including commanding them to perform specific actions;

  - Notification services that allow applications to publish/subscribe to broadcast messages (events) without explicit knowledge of their recipients/publishers;
  - Logging services that allow applications to record historically useful information; and
  - Alarm services that allow applications to broadcast alarm and health messages.

- *Persistence support* – These are services that allow applications to store and retrieve information whose lifetimes exceed that of a single instance of the application.
- *Tools* – These are libraries of software modules that are of common use across multiple system packages.
- *Application support* – This is support for writing ATST applications. The core implementation (i.e., **Component**) provides the connection framework to CSF. An extension (i.e., **Controller**) handles multiple, simultaneous configurations in a Command/Action/Response model. Either may be extended by developers to add specific functionality and subclasses are already provided to assist in the sequencing of actions and in real-time device control. ATST provides a Base Application Support Environment (**Base**) on top of CSF that provides this extended support through the **BaseComponent** and **BaseController**. *All ATST-specific applications are developed on top of either* **BaseComponent** *or* **BaseController**. The **Base** package provides a number of subclasses of **BaseComponent** and **BaseController** that support specific ATST functionality, including *motor controllers*, *sequencers*, and *device simulators*.

All CSF development is available for use in two languages, Java and C++, although the access to the services varies with the language. Scripting support, with full access to CSF is available from with Java applications using the **Jython** embedded interpreter.

2.3 DESIGN HIGHLIGHTS

Most of the design of the ATST Common Services Framework is of little interest to software development teams using the framework. However, a quick look at some of the key design features can be informative and also help illustrate some of the power and flexibility provided by CSF. Detailed information on the *use* of these, and other common services features, can be found in later sections of this document.

2.3.1 Communications-Neutral Architecture

While ATST has selected Internet Communications Engine (ICE) as the communications middleware that is the foundation for intra-application communications, CSF is designed to operate as independently as possible from the choice of *communication middleware*. The role of third-party middleware is carefully defined and bounded. This allows ATST to remain flexible on its choice of middleware, such as ICE, CORBA, or DDS, and to more easily replace one choice with another should it prove advantageous to do so in the future. Component developers should not be concerned with the choice of communications middleware –they reference no middleware-specific features, extend no middleware-specific classes, etc.

2.3.2 Separation of Functional and Technical Behavior

The ATST software design distinguishes between *functional* and *technical* behavior. Functional behavior describes the actions taken to directly implement ATST operations and can be contrasted with the technical behavior — the actions required of the infrastructure needed to support the functional behavior.

For example, logging a specific message into a persistent store is functional behavior — only the application developer can determine what (and when) messages should be logged. The underlying mechanism that performs the logging, however, is technical behavior. By establishing a clear distinction between functional and technical behavior, and providing the technical behavior through the ATST Common Services Framework, the application developer can concentrate on providing the required functionality.

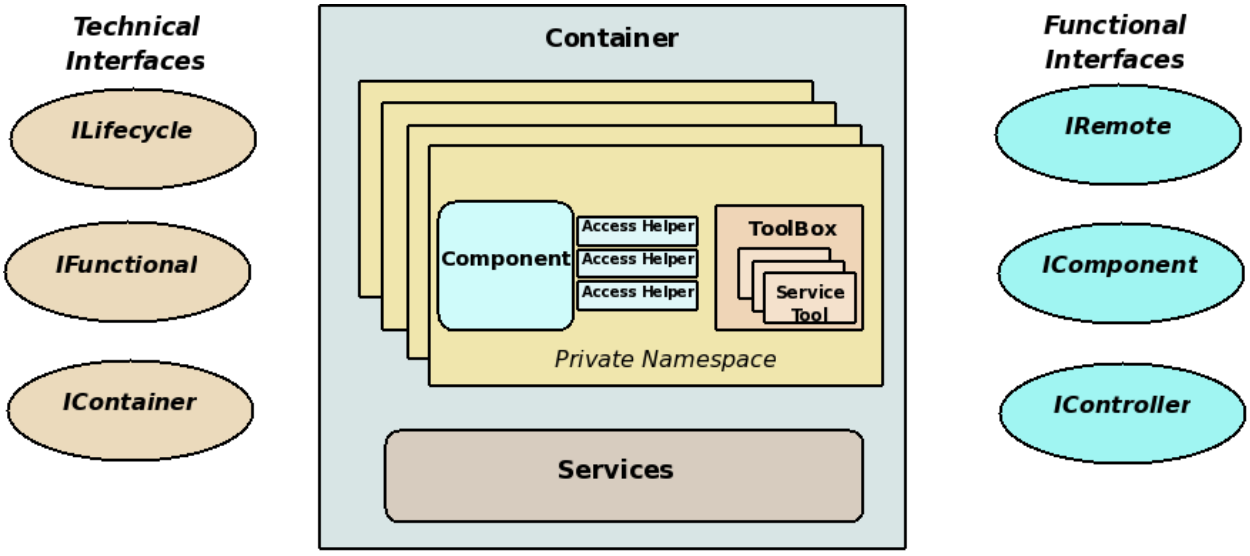
2.3.3 Configuration-Driven Control

A fundamental precept of the ATST software design is the use of *configurations* to drive ATST control behavior. A configuration is a set of logically-related, named values that describe the target condition of a subsystem. Control of a subsystem is accomplished by directing the subsystem to *match* the target conditions described by each configuration. The set of available commands is thus kept small and generic—little more than "match this configuration". Subsystems are responsible for determining how to match the target—all details of sequencing subsystem components are isolated in the subsystem.

Subsystems announce that they have met (or *cannot meet!*) the target using broadcast *events*, allowing fully asynchronous operation of components.

2.3.4 Container/Component Model

One feature of CSF is its adoption and support of a Container/Component Model (CCM). This approach, also used in the ALMA common services, is based upon the same fundamental design principles as Microsoft's .NET and Java's EJB architectures and simplifies application deployment and execution within a distributed environment. In the CCM, the deployment and lifecycle aspects of an application are separated from the functional aspects of the application. In particular, management applications (*containers*) are responsible for creating, starting, and stopping one or more functional applications (*components*). Containers are implemented as part of the common services, as are the base classes used by all components.



The *technical* interfaces (appearing on the left of the above diagram) are implemented within the common services framework by the underlying infrastructure. This means that developers can concentrate on providing code for performing actions visible through the *functional* interfaces (on the right of the above diagram). In the vast majority of cases this means sub-classing the **Controller** class, overwriting or

implementing any interface methods that access added functionality, and then writing support methods implementing that added functionality.

2.3.4.1 Components and Controllers

Components and controllers are the foundation for all applications. (A controller adds configuration management support to a component.) The bulk of the ATST software effort is designing and implementing the required functionality on top of components and controllers, extending either as appropriate. All ATST *functional behavior* is provided on top of **BaseComponent** and **BaseController**.

2.3.4.2 Containers

Containers provide a uniform means of deploying and controlling the technical aspects of component operations. Component developers can develop components without a detailed understanding of containers. Components also isolate individual components from each other by placing each loaded component into a private namespace while still allowing components shared access to common service implementations.

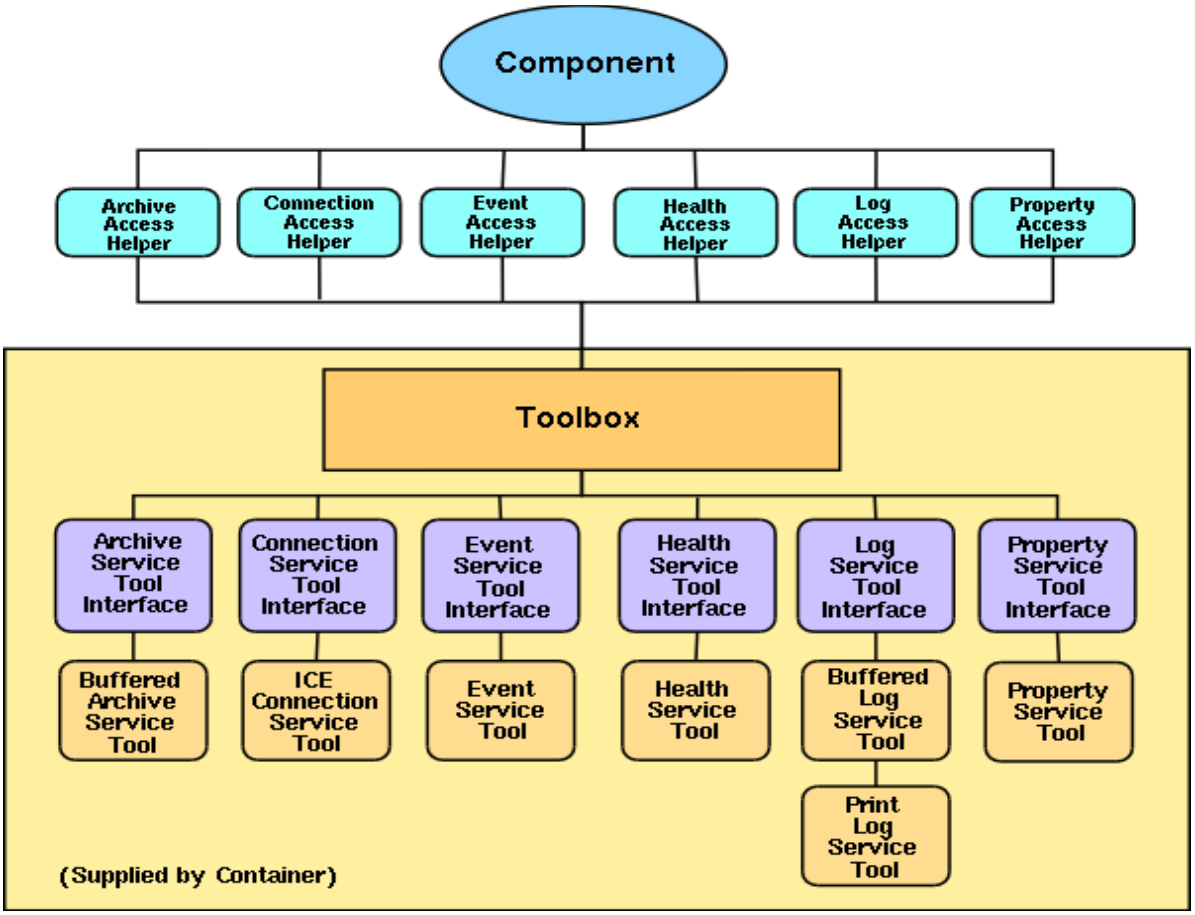
2.3.5 Service Toolboxes

In ATST, access to services is provided to components through the container. Some services may be shared among components while others might be unique to individual components. It is the container's responsibility to ensure that services are properly allocated. When a component is deployed to a container, the container assigns a unique service *toolbox* to that component, and places *service tools* into that toolbox. Service tools are modules that understand how to access specific common services. Typically, these tools are small and have well-defined tasks. However, the tools are designed to be *chained* so that several simple tools can be used to perform complex actions on service access. For example, message logging may be accomplished by chaining a log database tool that logs messages to a persistent store with a filter tool that looks for specific message characteristics. When a message with those characteristics is found, the filter tool might route that message to an operator's console. As a more extreme example (though not one likely to be used in ATST), it is possible to chain a connection service tool for ICE with a connection service tool for CORBA, allowing components to connect seamlessly and simultaneously to both ICE-aware and CORBA-aware modules! A container also retains access to each component's toolbox, permitting dynamic reconfiguration of tools without involving the component itself.

An important characteristic of the toolbox and service tools is that *all* component specific information needed by the various service tools is maintained in the toolbox, not in the specific service tool. This allows toolboxes to contain service tools that can be *shared* among components if it is advantageous to do so. For example, message logging may be more efficient if a common logging tool is shared among all the components within a container.

Toolboxes themselves are also chainable. This allows software tools specific to application layers installed on top of CSF to be added and managed by the Container/Component support built into CSF. For example, the *Application Base* layer that adds ATST-specific support to the technical architecture chains in a toolbox for ATST-specific services such as the *Header Service*.

While a component is capable of directly accessing the service tools in its toolbox, by far the most common way to access services is through static *service access helper* classes that are also provided by common services. These classes encapsulate access to the toolbox and its tools within easy to use static methods. It is this access through these access helpers that is discussed in detail in later sections of this document. *Direct access to service tools and the toolbox is intentionally not covered*.


Component
Archive Access Helper
Connection Access Helper
Event Access Helper
Health Access Helper
Log Access Helper
Property Access Helper
Toolbox
Archive Service Tool Interface
Connection Service Tool Interface
Event Service Tool Interface
Health Service Tool Interface
Log Service Tool Interface
Property Service Tool Interface
Buffered Archive Service Tool
ICE Connection Service Tool
Event Service Tool
Health Service Tool
Buffered Log Service Tool
Property Service Tool
Print Log Service Tool
(Supplied by Container)

3 INFRASTRUCTURE

3.1 COMMUNICATIONS

The ATST software system is very much a *distributable* system. This means that the communications infrastructure plays a critical role and is provided as part of the ATST Common Services Framework. Much of the implementation of the communications infrastructure is based upon services and support features found in third-party communications middleware packages. However, CSF isolates the dependence on third-party middleware from the rest of the ATST software system so that replacing the middleware is always a viable option.

Some of the key features of the CSF communications infrastructure are:

- Middleware isolation—as mentioned above, CSF isolates the third-party communications middleware from the rest of the ATST software.
- Multiple communication methods:
    - Peer-to-peer *command* messaging allows arbitrarily complex messages to be sent directly from one application to another.
    - Publish-subscribe *event* messaging allows generating and receiving messages by applications without regard to the intended message recipients or sources.
    - Bulk data *streaming* allows large data sets to be routed efficiently.
- Simple connection support—all that is needed for an application to establish a peer-to-peer connection to another application is the *name* of the target application. The communications infrastructure will locate the target application.
- Heartbeat monitoring—applications are watched and an alarm is raised if an application unexpectedly stops responding.

3.2 ATTRIBUTES

Attributes are the atomic representations of data items that can be communicated between ATST applications outside of the bulk data transport. An attribute is a (name, value) pair, where the *value* field is a sequence of strings. There is no limit to the size of this sequence or to the length of the individual strings. In practice, most attribute values have a single string field within the sequence. CSF provides support for converting between the strings used within attributes to hold values and the basic data types used by ATST.

While CSF itself imposes no restrictions on an Attribute's name field, ATST requires that all Attributes used in inter-application communication be uniquely named. The convention is to use the hierarchical structure of the ATST software element to produce *fully-qualified attribute names*, e.g., **atst.tcs.ecs.az.pos** names the enclosure azimuth position Attribute used by the Enclosure Control System contained within the Telescope Control System. CSF assists in automatically qualifying newly constructed Attributes based on the device that is referencing the Attribute. For example, an Attribute created inside of the Telescope Control System (**atst.tcs**) whose name is given simply as **pos** would be assigned the name **atst.tcs.pos**. Any Attribute whose name does not contain a '**.**' (period) is assumed to

be unqualified (*simple*) and is qualified based on the name of the application creating the Attribute. The Attribute name is left alone if there is a '**.**' in its name. The exception to this is that an attribute name that starts with a period will also be auto qualified.

Examples of attributes created inside of the telescope control system (**atst.tcs**):

`Attribute("pos")` ➔ attribute named "**atst.tcs.pos**"

`Attribute("atst.tcs.ecs.pos")` ➔ attribute named "**atst.tcs.ecs.pos**"

`Attribute(".ecs.pos")` ➔ attribute named "**atst.tcs.ecs.pos**"

The common services framework provides a limited form of automatic validity checking on Attribute values. Associated with every Attribute is a *metadata* description of that attribute. This metadata is maintained as *properties* in a persistent store by CSF and can be accessed by an application for use in validity checking and type conversions. See the Property Service for details.

Attributes that CSF uses for technical purposes are always prefixed with a double underscore '__'. As long as fully qualified attribute names are used this should never result in a collision. This does however mean that you should NOT name the top level component as double underscore.

3.2.1 Java Representation

The class **atst.cs.data.Attribute** defines an Attribute. Most references, however, use the matching *interface* **atst.cs.interfaces.IAttribute** with the following constructors and methods:

```
// create a new attribute with a name and no value
Attribute(String name);

// create a new attribute with a name and a value
Attribute(String name, {T} value);
// {T} may be one of: boolean, boolean[], Boolean, Boolean[], long,
// long[], Long, Long[], double, double[] Double, Double[], String,
// String[], Integer, Integer[], Float, or Float[].

//set the Attribute's value (see above note about {T}
void setValue({T} newValue);

// Get the value of the Attribute as the String
{T} get{TYPE}();
// TYPE may be one of: Boolean, BooleanArray, Float, FloatArray, Double,
// DoubleArray, Integer, IntegerArray, Long, LongArray, String,  or
// StringArray.  The return value will always be the Object type i.e.
// Boolean not boolean, or Long[] not long[]

// produce the Attribute's name
String getName();

// set the name of the Attribute
void setName(String name);

//print the attribute to standard output.
void show()
```

3.2.2 C++ Representation

The class **atst::cs::data::Attribute** defines an attribute. Most references, however, use the matching *interface* **atst::cs::interfaces::IAttribute.** Within CSF attributes and other data types that are passed around are generally passed around as smart pointers. Smart pointers provide help to prevent memory leaks.

```
typedef pIAttribute tr1::shared_ptr<IAttribute>;
pIAttribute Attribute::create();

pIAttribute Attribute::create(const string& name,
const {T}& value);
// {T} may be one of: bool, vector<bool>, int, vector<int>,
// int64_t, vecotr<int64_t>, double, vector<double>
// float, vector<float>, string, vector<string>

// Set(change) the name of the attribute
void setName(const string& newName);

// Get the name of the attribute
string getName() const;

// Set the value of the attribute
void setValue(const {T} newValue);
// See constructor details forinformation about {T}

// Return the value of the attribute
{T} get{TYPE}Value();
// TYPE may be one of: Boolean, BooleanArray, Float, FloatArray, Double,
// DoubleArray, Integer, IntegerArray, Long, LongArray, String,  or
// StringArray.  The return value will always be the Object type i.e.
// Boolean not boolean, or Long[] not long[]

//print the attribute to standard output.
void show()
```

3.3 ATTRIBUTE TABLES

Attributes can be grouped into *sets of Attributes* called *Attribute Tables*. Attribute tables are *unordered*, but can be searched efficiently. Note that this is a *set* and not a *multi-set*: at most one instance of an attribute with a given name may exist within an attribute table. Inserting a new attribute with the same name *replaces* the old one.

Attribute names are fully qualified (see above in the Attribute section). CSF attempts to help auto-qualify attributes inside of an attribute table. It does this based on the name of the enclosing class accessing the attribute table. For example, if the Telescope Control System (**atst.tcs**) attempts to retrieve an attribute named **pos** from a table, the table will return an attribute named "**atst.tcs.pos**". This accessing is done in the same way as the auto-qualifying is done in attribute creation.

Here are some examples of accessing an attribute in Enclosure Control System (**atst.tcs.ecs)**:

`someTable.get("pos")` ➔ gets an attribute named "**atst.tcs.ecs.pos**"

`someTable.get("atst.tcs.ecs.pos")` ➔ gets an attribute named "**atst.tcs.ecs.pos**"

`someTable.get(".az.pos")` ➔ gets an attribute named "**atst.tcs.ecs.az.pos**"

In ATST, the most common subclass of attribute table is the *Configuration*.

3.3.1 Java representation

The class **atst.cs.data.AttributeTable** defines an **AttributeTable** object. Most references, however, use the matching *interface* **atst.cs.interfaces.IAttributeTable** with the following basic methods:

```
// Construct a new Attribute Table with no elements
AttributeTable();

// Construct a new Attribute Table with the following elements
AttributeTable(java.util.Map<String, String[]>);

// does the table include this Attribute?
boolean contains(String attributeName);

// does the table include these Attributes?
boolean containsAll(String[] attributeNames);

// do any of the Attribute names in this table include this prefix?
boolean containsPrefix(String prefix);

// do any of the Attribute names in this table include this suffix?
boolean containsSuffix(String suffix);

// insert Attribute into this table
void insert(String name);
void insert(IAttribute attribute);
void insert(String name, IAttribute attribute);

// create and insert an Attribute into this table
void insert(String name, String… values);
void insert(String name, boolean… values);
void insert(String name, long… values);
void insert(String name, int… values);
void insert(String name, double… values);
void insert(String name, float… values);
void insert(String name, IAtstDate… values);

// remove (and return) an Attribute
IAttribute remove(String attributeName);

// produce an Attribute
IAttribute get(String attributeName);

// remove all entries from this table
void clear();

// remove all values from the entries in this table
void clearValues();
```

Additional methods are provided for convenience, including:

```
// number of Attributes in table
```

```
int size();

// produce the names of all Attributes
String[] getNames();

// produce the names of all Attributes sorted by name
String[] getSortedNames();

// remove all Attributes with names sharing a common prefix/suffix
// and return them in a new attribute table
IAttributeTable extractOn[Pre|Suf]fix(String [pre|suf]fix);

//same as extract on except the original table is unchanged
IAttributeTable selectOn[Suf|Pre]fix(String [pre|suf]fix);

// insert all elements in aTable
void merge(atst.cs.interfaces.IAttributeTable aTable);

// same as get(name).get{TYPE}() (see the attribute examples)
{T} get{TYPE}(String name);

// clone the attribute table and change the prefix of all of the
// attribute names from oldPrefix to newPrefix.
IAttributeTable requalify(String oldPrefix, std::String newPrefix);

// clones the attribute and change all of the
// auto-qualified prefixes into the new Prefix
IAttributeTable requalify(String newPrefix);
```

3.3.2 C++ representation

The class **atst::cs::data::AttributeTable** defines an **AttributeTable** object. As with attribute smart pointers are the preferred way of passing around attribute tables. Most references use the matching *interface* **atst::cs::interfaces::IAttributeTable** with the following methods:

```
// Create a new AttributeTable
pIAttributeTable AttributeTable::create();

// Does the table include this Attribute?
bool contains(const std::string& attributeName) const;

// Does the table include these Attributes?
bool containsAll(std::vector<std::string> aNames) const;

// Do any of the Attribute names in this table include this prefix?
bool containsPrefix(const std::string& prefix);

// Do any of the Attribute names in this table include this suffix?
bool containsSuffix(const std::string& suffix);

// Insert a new Attribute into the table
void insert(pIAttribute newAttribute);
void insert(const string& name);
void insert(const string& name, pIAttribute attribute);

// Create and ~~I~~insert a new Attribute into the table
```

```
void insert(const std::string& name, const std::string value);
void insert(const string& name, const vector<string> values);
void insert(const string& name, const bool value);
void insert(const string& name, const vector<bool> values);
void insert(const string& name, const int64_t value);
void insert(const string& name, const vector<int64_t> values);
void insert(const string& name, const int value);
void insert(const string& name, const vector<int> values);
void insert(const string& name, const double value);
void insert(const string& name, const vector<double> values);
void insert(const string& name, const float value);
void insert(const string& name, const vector<float> values);
void insert(const string& name, const pAtstDate value);
void insert(const string& name, const vector<pAtstDate> values);

// Insert a new Attribute into the table
void insert(const std::string& name, const std::vector<std::string>
      values);

// Remove (delete) an Attribute
pIAttribute remove(const std::string& attributeName);

// Retrieve an Attribute
pIAttribute get(const std::string& name) const;

// Remove all entries from this table
void clear();

// Remove all values from the entries in this table
void clearValues();


// Return the number of Attributes in the table
int size() const;

// Return the names of all Attributes
std::vector<std::string> getNames() const;

// remove all Attributes with names sharing a common prefix/suffix
// and return them in a new attribute table
pIAttributeTable extractOn[Pre|Suf]fix(const std::string& [pre|suf]fix);

//same as extract on except the original table is unchanged
pIAttributeTable selectOn[Suf|Pre]fix(const std::string& [pre|suf]fix)
      const;

// Insert all elements from source table
void merge(const pIAttrbute source);

// Return stringified table
std::string toString() const;

// Display table to standard output
void displayAttributes() const;

// Display table to standard output with heading
void show(const std::string heading) const;
```

```
// same as get->get{TYPE}(); (see the attribute examples)
{T} get{TYPE}(const std::string& name) const throw(TraceableException);
```

3.4 CONFIGURATIONS

While attributes are the atomic unit of CSF communications, the fundamental data structure for submitting instructions to controllers is the *Configuration*, an extension of the **AttributeTable** class. Configurations are used in ATST to describe the conditions that must be met by the target controller. Controllers report back their success or failure in matching those conditions.

Configurations add a few pieces of information to an **AttributeTable**; **configId, sourceId, creationTimeStamp, and ancestry**. The **configId** is a unique identifier that allows the CSF technical architecture to track configurations as they are passed to controllers and processed by controllers. After submitting a configuration to a controller the configuration's id may be passed to a controller to instruct the controller to pause, resume, cancel, or abort the processing of the configuration. The **sourceId** is used to identify the application that created a particular configuration. This is generally only used for internally. The **creationTimeStamp** is used to identify when a configuration was created. This is generally only used internally. Ancestry records Configuration lineage allowing one to trace how any given Configuration came to be in its current state. In order to preserve ancestry Configurations should be created using methods and constructors that build upon a previous Configuration.

Being that configurations are extensions of attribute tables, the same rules regarding attribute names exits. Namely the attribute names should always be fully qualified, and the double underscore '__' prefix should be considered reserved for the technical architecture.

When the processing of one configuration leads to the construction of a new (derived) configuration, the derived configuration's id should be related to the base configuration's id.

In addition to the methods described below, all methods inherited from attribute table that returned attribute tables now return configuration. This includes methods like **extractOnPrefix()** and **selectOnSuffix()**. The returned configuration's id is derived from the base configuration's id.

3.4.1 Java representation

```
// constructor for a new configuration
// this constructor is for special use only
Configuration();

//constructor for a derived config
Configuration(IConfiguration baseConfig);

//constructor for a derived config with a map of attributes
Configuration(IConfiguration baseConfig, Map<String, String[]> map);

//get the config id
String getId();

//get the cofinguration ancestry
```

String getAncestry();

3.4.2 C++ representation

```
//factory constructor for a new configuration
// this factory constructor is for special use only
static pIConfiguration create();

//factory constructor to create a derived config
static pIConfiguraiton create(pIConfiguraiton baseConfig);

//get the config id
std::string getId();

//get the cofinguration ancestry
std::string getAncestry();
```

3.5 DATA EXAMPLE

Below are some simple code examples that show some basic usage of attributes, attribute tables, and configurations.

3.6 JAVA EXAMPLE

```
// create three attributes two with strings and one with an int
IAttribute attribute1 = new Attribute("atst.tcs.mode", "track");
IAttribute attribute2 = new Attribute("atst.tcs.trackid", 123456);
IAttribute attribute3 = new Attribute("atst.ics.instrument", "visp");


// craete an attribute table
IAttributeTable table = new AttributeTable();

// insert the attributes into a table
table.insert(attribute1);
table.insert(attribute2);
table.insert(attribute3);


// create a configuration whose unique id starts with 'example'
IConfiguration config1 = new Configuration("example");
// put all of the attribute from the table into the configuraiton
config1.merge(table);

// generte a new config derived from config1 that contains only
// the attributes intended for the tcs
IConfiguration config2 = config1.selectOnPrefix("atst.tcc");
if(config1.contains("atst.tcs.mode"){
   System.out.println(config1.get("atst.tcs.mode").getString());
   //or the equivolent
   System.out.println(config1.getString("atst.tcs.mode"));
}

//create a configuration derived from config1
IConfiguration config3 = new Configuration(config1);
//put all of the attribute from the table that should go to
//the ics into the configuraiton
```

```
config3.merge(table.selectOnPrefix("atst.ics"));
```

3.7 C++ EXAMPLE

```
// create three attributes two with strings and one with an int
pIAttribute attribute1 = Attribute::create("atst.tcs.mode", "track" );
pIAttribute attribute2 = Attribute::create("atst.tcs.trackid", 123456 );
pIAttributep attribute3 = Attribute::create("atst.ics.instrument", "visp"
      );


// craete an attribute table
pIAttributeTable table = AttributeTable::create();

// insert the attributes into a table
table->insert(attribute1);
table->insert(attribute2);
table->insert(attribute3);


// create a configuraiton whose unique id starts with 'example'
pIConfiguration config1 = Configuration::create("example");
config1->merge(table);

// generte a new config derived from config1 that contains only
// the attributes intended for the tcs
pIConfiguration config2 = config1->selectOnPrefix("atst.tcc);
if(config1->contains("atst.tcs.mode"){
   std::cout << config1->get("atst.tcs.mode")->getString();
   //or the equivolent
   std::cout << config1->getString("atst.tcs.mode");
}

//create a configuration derived from config 1;
pIConfiguration config3 = Configuration::create(config1);
//put all of the attribute from the table that should go to
//the ics into the configuraiton
config3->merge(table->selectOnPrefix("atst.ics"));
```

3.8 COMMANDS

The two fundamental mechanisms for communication between ATST components are *Commands* and *Events*. Commands are used in peer-to-peer communication while events are used in publish/subscribe communication.

There are two basic classes of commands used in ATST:

- Lifecycle commands – commands used by ATST system management to control the *lifecycle* characteristics of applications. Users generally do not need to be concerned with the lifecycle commands because they are implemented by the underlying ATST infrastructure. However, they are introduced in the section on Containers.

- Functional commands – commands that implement the specific functional characteristics of a Component. Because the ATST uses configurations and a narrow command interface, the number of functional commands is quite small.

The majority of ATST functional operation is based on the *Command/Action/Response* model that isolates the transmission of a configuration from the resulting action that is performed. When an application receives a configuration, it validates that configuration and immediately *accepts* or *rejects* the configuration. If the configuration is accepted, the application then schedules an independent internal action to meet the conditions imposed by the configuration. Once those conditions have been met, an *event* is posted signifying the successful completion of the action (or the unsuccessful completion if the conditions cannot be met).

The functional commands depend upon the type of each component and are covered in detail in the sections on components and controllers.

3.9 EVENTS

Events are the basis of the publish/subscribe communications system provided by CSF. Any application may post events and/or subscribe to events posted elsewhere. The CSF event service is robust and high performance.

An event consists of a **name** and a **value**. A sequence of events with the same name is referred to as an *event stream*. The name of the event is used to identify the event stream to subscribers. The value is an arbitrary attribute table. The convention is that all events in a stream share the same structure (i.e., the same attribute names within the attribute table).

The event service has the following general properties:

- An event stream represents a many-to-many mapping: events may be posted into the stream from more than one *source* and received by zero or more *targets*. (Typically, however, most event streams will have a single source.)

- Events posted by a single source into an event stream are received by all targets in the same order as they were posted.

- Delivery of events to one subscriber cannot be blocked by the actions of another subscriber.

- Events are not queued by the service. A "late" subscriber will not see earlier events.

- The event service does not *drop* events. A published event will be delivered to all subscribers. The event service supports arbitrary event names. However, ATST itself imposes a *hierarchical* naming convention on event names.

- Events are *automatically* tagged with the source and a timestamp.

See the Event Service section for details.

3.10 CONTAINERS AND COMPONENTS

ATST software is based on the *Container/Component Model*. In this model, one or more components are deployed into each container. There are separate containers for Java and C++ components. Typically there is one container per host but there is no requirement restricting containers in this fashion.

Containers are responsible for deploying components, providing those components with access to the CSF services and tools, and monitoring the health of components. This approach means that there is a uniform method for managing components across the ATST system. Furthermore, component developers may safely ignore the majority of lifecycle characteristics and can focus on the development of the specific functionality required of each component.

ATST Containers also provide each Component with access to the services. A container creates a separate **ToolBox** class for each component and then populates that Tool Box with the *service tools* that provide access to the common services. The Tool Box allows the container to retain access to the service tools so the container can, if needed, dynamically adjust this access (for example, the service tool supporting event receipt by a subscribing component could be adjusted to log received events during debugging). Tool Boxes also provide a standard location for service tools to hold commonly used information about the component.

When a container creates a component, it provides that component with a *private* namespace that is separate from that of other components in that container; component developers do not have to be concerned about namespace collisions with other components. Service tools, on the other hand, may be loaded either within each component's private namespace (i.e., a separate copy of the service tool is needed for each component) or in a *shared* namespace (i.e., one service tool may used by multiple components in the container). The container determines which service tools are private and which are shared. The choice is completely transparent to the components.

To be managed by a container, all components must adhere to a common standard. This standard is implemented by the base component class and is discussed in detail in the sections on components and controllers. A quick overview is presented below.

3.10.1 Deploying a component

While the details aren't particularly important to component developers, it is instructive to see the major steps that a container takes when deploying a component. A container automatically deploys the components that it contains when the container transitions into the running state. When deploying components a container:

1. Loads the component's class into a private namespace;

2. Creates a toolbox in that same namespace, populated with tools, and attached to the component

3. Instantiates the components by invoking its *default* constructor in a new namespace

Note that the functional operation of the component is not started as part of the deployment. This allows multiple components to be deployed and configured before any are started operating. Also note that access to CSF services is *available* to the default constructor. However, the default constructor for a component should do as little as possible, deferring any initialization actions until the **init** method.

3.10.2 Managing a component's lifecycle

Components respond to lifecycle commands that drive them through their lifecycle stages.

Initializing a component

Deployed components are first initialized. This stage allows component developers an opportunity to prepare for startup. A component may configure internal storage, connect to other components, and perform other basic configuration tasks during initialization. The expectation is that the component's **init** method is a *software-only* operation – no mechanisms should move as a result of calling **init**. Furthermore, the component is not assumed to be ready to operate upon completion of the **init**.

Starting a component

At some point after the initialization of a component, the component may be instructed to start running by invoking the component's **startup** method. After startup the component is considered *running* and ready to accept functional commands.

Stopping a component

A running component may be instructed to stop running by invoking the component's **shutdown** method. The **shutdown** method is the logical inverse of the **startup** method. Once a component has been shutdown, it can only be restarted with the **startup** method or removed from the system.

Un-initializing a component

A stopped component may be un-initialized by invoking its **uninit** method. The uninitialized stage gives the component developer a place to release resources that were acquired during the initialization stage.

Removing a component

The following major steps are taken when a component is to be removed from the system:

1. The container verifies that the component has been shutdown.
2. The container accesses the component's Tool Box and releases all service tools.
3. The container removes the component's instance.

Components are automatically removed from their container when the containers transitions out of the running state.

3.11 SERVICES

The services provided by CSF can be loosely divided into two categories:

- The **major** services (also simply referred to simply as *services*) provide functionality that is required for the successful operation of *any* component. The major services are:
    - The connection service provides support for connecting components and controllers with each other including uniform access to ATST's *Command/Action/Response* system.

  - The event service provides access to ATST's publish/subscribe communications system.
  - The logging service provides a standard means of logging many types of messages.
  - The health service provides uniform handling of system error conditions.
  - The property service provides persistent storage for default attributes.
  - The alarm service provides a way to notify operators of issues requiring prompt attention

- The **minor** services (all referred to as *tools*) provide support useful for implementing functionality within components. They are also used internally to implement the major services. The minor services are discussed in the section on Tools. (Tools in this context should not be confused with *service helper tools* used to implement many services found in a component's Tool Box.)

Access to both services and tools by components is simplified through the use of *helper* classes that mask most of the details of the services from the component developer. These helper classes are covered in detail in the following sections. The actual implementation of service access using the Tool Box and service helper tools is deliberately *not covered*.

The following sections present introductions to the use of each of the services. The full documentation on the service APIs is found in the ATST source code documentation.

4 CONNECTION SERVICE

The connection service supports component registration with the ATST communications system to allow other components direct *peer-to-peer* access. The service also supports the connection to other components. The following basic actions are provided by the connection service:

- Registration – the current component is registered by name with the ATST communications system.
- Deregistration – the current component's registration is removed.
- Connection – the current (*source*) component connects to another (*target*) component using the name of the target.
- Disconnection – the current component's connection to the named target is removed.

Registering and unregistering components are handled automatically by the ATST Common Services Framework. Component developers do not need to perform these two actions in their code.

4.1 COMMANDS

Once a connection has been made by a source component to a target component, the source component may issue *commands* to the target component. (*Whether or not those commands are accepted by the target component depends upon the **access policies** in effect.*) The specific commands available on the target component depend upon the interface selected as part of the connection to the target. The available interfaces are:

- **IRemote** – the minimal remotely accessible object in the Common Services Framework
- **IComponent** – a simple target **Component**.
- **IController** – a target controller (a subclass of **IComponent**).

The **IComponent** and **IController** interfaces are selected by *downcasting* the **IRemote** object produced by connecting to a target component. Only legal downcasts are allowed, for example, a non-controller target connection cannot be downcast to **IController**.

All commands in the **IRemote** and **IComponent** interface respond synchronously. The **IController** interface introduces asynchronous action responses to the command: **submit(configuration)**. A submitted configuration initiates a separate asynchronous action with a controller. The configuration is queued by the controller and the resulting action can be scheduled to perform immediately or in the future. The **submit** command itself returns immediately without waiting for the response.

All submitted configurations are automatically tagged with a *header tag*. This tag identifies that the configuration is a result of some behavior driven by a specific observation. The targeted controller remembers this tag and compares it with subsequent *header collection events* to determine if a header event needs a response.

4.2 CONTROLLER ACTIONS

Commands return immediately but the actions that are initiated as a result of a submit command may take some time to complete. When the action completes, an *action status event* is posted that includes the completion status of that action. The component generating the command initiates the monitoring of this status event prior to issuing the command on the remote system. This monitoring is performed automatically by CSF but controller developers need to attach a *callback* to perform processing on action completion. This callback may be **null** if no further processing is needed.

The status event's name is prefixed with the name of the Component posting the event, as is the name of the *status Attribute* contained in that event's value. For example, if the Controller **atst.tcs.mcs** posts an action status event, the name of that event is **atst.tcs.mcs.status** that matches the name of the action status attribute within the event's value.

The details of the available commands for each of these interfaces are language specific and covered in the appropriate *helper* section below.

4.3 JAVA HELPER

The class **atst.cs.services.App** provides Java-based components with access to the connection service. The static methods involved are:

```
atst.cs.interfaces.IRemote App.connect(String targetComponentName);

void App.disconnect(String targetComponentName);
```

Note that **App.connect** returns an **IRemote**. The object that is returned can be *downcast* to the same correct interface. So, for example if **atst.tcs.mcs** is the name of a controller registered with the naming service then a component may connect to **atst.tcs.mcs** with:

```
atst.cs.interface.IController mount =
 (atst.cs.interface.IController)App.connect("atst.tcs.mcs");
```

Inappropriate downcasts result in a **ClassCastException**.

The interfaces available for downcast are:

- **atst.cs.interfaces.IComponent** – a simple target component
- **atst.cs.interfaces.IController** – a target controller

The **atst.cs.services.App** class provides an additional static method that returns the (registered) name for this application**.**

```
String getName();
```

4.3.1 Java commands for IRemote

The following command methods are defined by the **atst.cs.interfaces.IRemote** interface and so are available to both components and controllers:

```
void set(atst.cs.interfaces.IAttributeTable attributes);
```

```
atst.cs.interfaces.IAttributeTable
         get(atst.cs.interfaces.IAttributeTable attributes);
```

The **set()** method passes a set of attributes to the target component which uses this set to adjust its internal parameters. Typically, a target component ignores attributes that have no meaning to the component.

The **get()** method accepts a set of attributes with empty or irrelevant values – the return value is the same set of attributes with the values adjusted by the target component. Attributes that are unknown to the target component are left unchanged.

4.3.2 Java commands for IController

The **atst.cs.interfaces.IController** interfaces extends **IComponent** with the following methods:

```
void submit(atst.cs.interfaces.IConfiguration config,
    atst.cs.interfaces.IActionCallback callback);

// A configID of "*" will cancel all active actions.
void cancel(String configID);

// A configID of "*" will abort all active actions.
void abort(String configID);

void pause(String configID);

void resume(String configID);
```

4.3.3 Java action callbacks

When a remote controller completes the action initiated with a call to the **IController** interface method **submit**, the connection service receives the completion status event from the remote controller and invokes the appropriate action call back method. The **atst.cs.controller.ActionCallbackAdapter** implements the **atst.cs.interfaces.IActionCallback** interface and expects a developer to implement the following methods:

```
// the action has completed successfully
void doDone(atst.cs.interfaces.IConfiguration newConfig);

// the action stoped before completion (error, canceled, aborted)
void doAbort(atst.cs.interfaces.IConfiguration newConfig,
         String reason);

// the action posted a status report
void doReport(atst.cs.interfaces.IConfiguration newConfig);

// the action has been inserted into the controlle'r action queue
void doScheduled();

// the action is now running (no longer paused or no longer scheduled)
void doRunning();

// the action is now paused
void doPaused();
```

Developers should implement these methods, adding any operations they need to have performed on action completion, or status reports.

4.3.4 Java example

The following example demonstrates how a developer might use the connection service. Note that the current implementation of the ATST communications system requires the Internet Communication Engine (ICE) and its companion services. Consult the Reference Manual for instructions on starting the ICE services.

Assume that a subclass of component exists on a remote machine and has been registered with the connection service. Registration of a component with the connection service happens automatically when it is loaded into a container. Then using the component's name, one can obtain a reference to the component and invoke operations on it as if it was a local object.

```
// Obtain a reference to the component and downcast appropriately
String componentName = "system.subsystem.device"
atst.cs.interfaces.IComponent component =
(IComponent)App.connect(componentName);

// Invoke operations
try {
   // Construct a list of status items to report on -- position, rate, ...
   IAttributeTable status = new AttributeTable();
   status.insert(new Attribute("system.subsystem.device.position", ""));
   status.insert(new Attribute("system.subsystem.device.rate", ""));

   // Obtain the values from the device
   IAttributeTable list = component.get(status);
   list.show("device status is: ");
} catch(Exception e) {
   e.printStacktrace();
}
```

4.4 C++ HELPER

The class **atst::cs::services::App** provides C++ based components with access to the Connection service. The static methods involved are:

```
pIRemote App::connect(const string& targetComponentName);

void App::disconnect(const string& targetComponentName);
```

Note that **App::connect** returns a smart pointer to an **IRemote** which can be downcast to the correct interface (i.e., **IController**). So, for example, if **atst.tcs.mcs** is the name of a controller registered with the connection service then a Component may connect to **atst.tcs.mcs** with:

```
pIRemote mcsRmt = App::connect("atst.tcs.mcs");
if ( !mcsRmt ) {
   cout << "error connection to atst.tcs.mcs";
   return;
}
pIController mcsCtrl;
mcsCtrl = std::tr1::dynamic_pointer_cast<IController, IRemote>(mcsRmt);
```

In either example, if **App::connect** fails to connect to the desired target component the returned pointer is set to zero.

The interfaces available for downcast are:

```
// a simple target Component
<atst::cs::interfaces::IComponent>;

// a target Controller
<atst::cs::interfaces::IController>;
```

The **atst::cs::services::App** class provides an additional static method that is useful to component developers:

```
// Returns the (registered) name for this application
string App::getName();
```

4.4.1 C++ commands for IRemote

The following command methods are defined by the interface **atst::cs::interfaces::IRemote**:

```
pIAttributeTable get(pIAttributeTable attributes);

void set(pIAttributeTable attributes);
```

The **set** method passes a set of attributes to the target component which uses this set to adjust its internal parameters. Typically, a target component ignores attributes that have no meaning to the component.

The **get** method accepts a set of attributes with empty or irrelevant values. The return value is the same set of attributes with the values adjusted by the target component. Attributes that are unknown to the target component are left unchanged.

4.4.2 C++ commands for IController

The **atst::cs::interfaces::IController** interface extends **IComponent** with the following methods:

```
void submit(pIConfiguration config, pIActionCallback callback);

// A configID of "*" will cancel all active actions.
void cancel(String configID);

// A configID of "*" will abort all active actions.
void abort(String configID);

void pause(String configID);

void resume(String configID);
```

4.4.3 C++ commands for callbacks

When a remote controller completes the action initiated with a call to the **IController** interface method **submit**, the connection service receives the completion status event from the remote controller and invokes the appropriate action callback method. The **atst::cs::controller::ActionCallbackAdapter**

implements the **atst::cs::interfaces::IActionCallback** interface and expects a developer to implement the following methods:

```
// the action has completed successfully
void doDone(pIConfiguration newConfig);

// the action stoped before completion (error, canceled, aborted)
void doAbort(pIConfiguration newConfig, const std::string& reason);

// the action posted a status report
void doReport(pIConfiguration newConfig);

// the action has been inserted into the controlle'r action queue
void doScheduled();

// the action is now running (no longer paused or no longer scheduled)
void doRunning();

// the action is now paused
void doPaused();
```

Developers should implement these methods, adding any operations they need to have performed on action completion, or status reports.

4.4.4 C++ example

The following example demonstrates how a developer might use the connection service. Note that the current implementation of the ATST communications system requires the Internet Communication Engine (ICE) and its companion services. Consult the Reference Guide (SPEC-0022-2) for instructions on starting the ICE services.

Assume that a subclass of component exists on a remote machine and has been registered with the connection service. Registration of a component with the connection service happens automatically when it is loaded into a container. Then using the component's name, one can obtain a reference to the component and invoke operations on it as if it was a local object.

```
std::tr1::shared_ptr<IRemote> mcsRmt = App::connect("atst.tcs.mcs");
if ( !mcsRmt ) {
   cout << "error connection to atst.tcs.mcs";
   return;
}
pIController mcsCtrl;
mcsCtrl = std::tr1::dynamic_pointer_cast<IController, IRemote>(mcsRmt);
pIAttribute pos = Attribute::create("atst.tcs.mcs.pos");
pIAttribute rate = Attribute::create("atst.tcs.mcs.rate");
pIAttributeTable status = AttributeTable::create();
status->insert(posn);
status->insert(rate);
pIAttributeTable list;
if (mcsCtrl) {
   list = mcsCtrl->get(status);
   list->show("atst.tcs.mcs status");
}
```

5 EVENT SERVICE

5.1 EVENTS

Publish/subscribe messaging is provided through the ATST *event service*. The event service allows components to post messages and to perform actions upon the receipt of messages, both without having to connect directly to other components. The event service provides, through a *helper* class, support for these basic operations.

Events themselves are simple *name/value* pairs. The *event name* is a single word (i.e., no spaces), typically written using camel case notation. The event value is an **AttributeTable**, but the event service helper class provides convenience methods for automatically embedding other types within that **AttributeTable**.

Events are received by attaching a callback to a subscription. The event service, upon receipt of an event, invokes this callback in a separate thread. However, *all events* received from the same subscription use the same thread so delivery order is preserved within the callback processing. If events are being received faster than the callback processing, the unprocessed events are normally locally queued within the event service. *This is a potential problem*, but represents a trade-off of mutually exclusive goals. Component developers are encouraged to write callbacks that process quickly. Numerous approaches are available to handle the case where the required action cannot be performed quickly - the best approach to use is dependent upon the nature of the specific task and is thus the responsibility of the component developer.

5.2 EVENT STRATEGIES

It is possible to use different strategies to publish or to subscribe to events. These strategies can increase throughput at the cost of latency, or help eliminate problems with events coming in faster than they can be processed.

5.2.1 Posting Strategies

The posting strategy selected (by default or otherwise) on the first post of an event is used for *all* subsequent posts of that event, regardless of future suggestions. The only way to change the active strategy is to reload the component. There are two event posting strategies currently available (as with the event handling strategy, not all strategies may be provided by all event service tools:

- **Event.POSTSTD** – suggests using the standard event post strategy. This mode *guarantees* event order delivery. This is normally the default posting strategy and is *always* available.

- **Event.BATCH** – suggests using a *batch* event delivery mode. This may not be available with all event services, but is expected to provide much higher throughput if it is. However, this mode impacts *latency* and so must be used carefully.

5.2.2 Subscription Strategies

Not all event service tools implement all strategies, so a suggestion for an unimplemented strategy is silently defaulted to an implemented one.

Currently, there are three strategies available:

- **NORMAL** – the processing of the event callback is synchronized with event receipt. If there are multiple subscribers to the same event, event delivery is effectively serialized at a rate no faster than the slowest callback handler using this strategy.

- **FAST** – the processing of the event callback is decoupled from event receipt, although the order of event delivery is maintained. The event poster is blocked a minimal amount of time on each delivery. This is particularly well suited to the case where there are multiple subscribers or if the callback is performing some complex task that might cause a timeout of the post operation if it had to block for that entire time. A subscriber that is unable to keep up with the rate of event delivery may consume large amounts of memory while queuing delivered events. **This is the most common default strategy.**

- **LOSSY** – similar to **Event.FAST** except that unprocessed delivered events are discarded as new events arrive. This avoids the memory growth possible in **Event.FAST** but means that callbacks using this strategy *must* be able to handle the fact that some events may not be delivered to it. Events are not processed faster than a specified rate (see below). The default rate is 0.1ms (100,000ns).

The event handling strategy may be modified with an optional *delay*. This delay has no meaning with **Event.NORMAL**. With **Event.FAST** the delay may be used to fine-tune *performance characteristics*, exchanging storage for speed. This is rarely needed. With **Event.LOSSY**, however, this delay can be used to *throttle* event callback handling. This is useful, for example, when displaying event values on a user interface where the desired screen update rate is considerably less than the event delivery rate. The delay can be specified as:

```
"["+delay_ms+"]"
```

or as with the Java **Object.wait()** methods:

```
"["+delay_ms+","+delay_ns+"]"
```

Delays less than 1000ns are silently ignored, as are format errors. This example could be used to process events no faster than 10Hz (100ms).

```
Event.LOSSY+"[100]"
```

5.3 EVENT POSTING

5.3.1 Java helper

The class **atst.cs.services.Event** provides Java-based Components with access to the event service. The static constants and methods provided by this class are:

```
static final String POSTSTD; //the standard posting strategy
static final String BATCH; //the batch posting strategy

//post an event with the given value
void post(String eventName, IAttributeTable value);
void post(String eventName, IAttribute value);

//post an event with the given name and an attribute with the given value
void post(String eventName, String attributeName, long value);
```

```
void post(String eventName, String attributeName, double value);
void post(String eventName, String attributeName, String value);
```

The **post()** methods also deserve some explanation. *Internally* the ATST event service uses just the **IAttributeTable** value on all posted events. This is the first form of **post** shown above. The other **post** methods convert the value passed into this internal representation. In the case of long, double, and String, the resulting **IAttributeTable** contains a single **IAttribute** with the supplied attribute name. In the case of value parameter being an **IAttribute** (the last of the above methods), the attribute is wrapped into an **IAttributeTable**. See the section on Event callbacks for a discussion on the support available when receiving such events.

There is a second form for the attribute and attribute table forms of the **post** methods with an additional "strategy" argument:

```
void post(String eventName, atst.cs.interfaces.IAttributeTable value,
      String strategy);
void post(String eventName, atst.cs.interfaces.IAttribute value, String
      strategy);
```

5.3.2 C++ helper

The class **atst::cs::services::Event** provides Java-based Components with access to the event service. The static constants and methods provided by this class are:

```
const static std::string POSTSTD; //the standard posting strategy
const static std::string BATCH; //the batch posting strategy

//post an event with the given value
void post(std::string& eventName, pIAttributeTable value);
void post(std::string& eventName, pIAttribute value);

//post an event with the given name and an attribute with the given value
void post(std::string& eventName, std::string& attributeName, long value);
void post(std::string& eventName, std::string& attributeName, double
      value);
void post(std::string& eventName, std::string& attributeName, std::string&
      value);
```

The **post()** methods also deserve some explanation. *Internally* the ATST event service uses just the **IAttributeTable** value on all posted events. This is the first form of **post** shown above. The other **post** methods convert the value passed into this internal representation. In the case of long, double, and String, the resulting **IAttributeTable** contains a single **IAttribute** with the supplied attribute name. In the case of value parameter being an **IAttribute** (the last of the above methods), the attribute is wrapped into an **IAttributeTable**. See the section on Event callbacks for a discussion on the support available when receiving such events.

There is a second form for the attribute and attribute table forms of the **post** methods with an additional "strategy" argument:

```
void post(std::string&eventName, pIAttributeTable value,
      std::string&strategy);
void post(std::string&eventName, pIAttribute value, std::string&strategy);
```

5.4 EVENT CALLBACKS AND SUBSCRIPTIONS

Components wishing to receive events perform actions on the basis of those events by attaching a callback object to the event subscription. Arbitrary actions may be performed by this callback object (but see the caveat in the first section on Events).

5.4.1 Java Helper

Component developers need to inject *functional* behavior into event callbacks. CSF provides an adapter class **atst.cs.services.event.EventCallbackAdapter** that must be overridden by component-specific code. Developers should override the single method:

```
public void callback(String eventName)
```

This is invoked by ATST's event service when a subscribed-to event is received. Note that the actual event value is not passed in as a parameter to this method. Instead, helper methods are provided by **EventCallbackAdapter** to obtain the value. For a full list of these helper methods see the API documentation. A few of these helper methods are:

```
// produce the named Attribute's value
String getString(String attributeName);
long getLong(String attributeName);
double getDouble(String attributeName);

// produce the event's full value as an AttributeTable
IAttributeTable getAttributeTable();
```

To subscribe the callback to an event stream you should use one of the methods provided in the event helper class: **atst.cs.services.Event**.

```
//subscribe the callback to the event stream using the given strategy
void subscribe(String eventName, IEventCallbackcallback callback, String
      strategy);

//same as subscribe(eventName, callbck, Event.NORMAL)
void subscribe(String eventName, EventCallbackcallback callback);

//unsubscribe the callback from the given event stream
void unsubscribe(String eventName, IEventCallbackcallback callback);

static final String NORMAL; // the 'normal' subscription strategy
static final String LOSSY; // the 'lossy' subscription strategy
static final String FAST; // the 'fast' subscription strategy
```

5.4.2 Java Example

The following Java code fragments demonstrate how one might use the event service to subscribe/publish events to a named event stream.

MyEventListener overrides the callback method of the **EventCallbackAdapter** class. This class demonstrates how one would deduce the data type of the attribute named after the event. In practice, the subscriber would expect a specific data type and would only call the conversion method appropriate for that data type.

```
import atst.cs.services.event.EventCallbackAdapter;

public class MyEventListener extends EventCallbackAdapter {
         public void callback(String eventName) {

       // Identify who sent the event
       System.out.println("event '"+eventName+"' received");

       // Is it a long value?
       Long lValue = getLong(eventName);
       if (null != lValue) {
          System.out.println("long value is: "+lValue);
          return;
       }

       // Is it a double value?
       Double dValue = getDouble(eventName);
       if (null != dValue) {
          System.out.println("double value is: "+dValue);
       return;
       }

       // Finally, is it a string value? (Must be null if not!)
       String sValue = getString(eventName);
       if (null != sValue) {
          System.out.println("string value is: "+sValue);
          return;
          }

       System.out.println("Value of '"+eventName+"' not found in event!");
          }
   }
```

The following code fragment demonstrates how an application would subscribe to a named event stream using an instance of **MyEventListener** as the remote callback object.

```
MyEventListener listener = new MyEventListener();
Event.subscribe("system.subsystem.device.status", listener);

   System.out.println("Press Q to quit");
   try {
      int ichr = 0;
      do {
         ichr = System.in.read();
      } while( (ichr != 'Q') && (ichr != 'q'));
}
catch(java.io.IOException ioexcept) {
   System.err.println("java.io.IOException occurred");
      ioexcept.printStackTrace();
}

System.out.println("unsubscribing");
Event.unsubscribe("system.subsystem.device.status", listener);
```

The following code fragment demonstrates how one would publish events of each supported data type to a named event stream.

```
String eventName = "system.subsystem.device.status";

// Post strings
System.out.println("posting strings ...");
for (int i = 0; i < 100; i++) {
        Event.post(eventName, "value", ""+i);
}

// Post doubles
System.out.println("posting doubles ...");
for (double d = .1; d < 10.0; d+=.125) {
        Event.post(eventName, "value", d);
}

// Post longs
System.out.println("posting longs ...");
for (long l = -100000; l < 100000; l+=5000) {
        Event.post(eventName, "value", l);
}
```

5.4.3 C++ Helper

Component developers need to inject *functional* behavior into event callbacks. CSF provides an adapter class **atst::cs::services::event.EventCallbackAdapter** that must be overridden by component-specific code. Developers should override the single method:

```
public void callback(const std::string& eventName)
```

This is invoked by ATST's event service when a subscribed-to event is received. Note that the actual event value is not passed in as a parameter to this method. Instead, helper methods are provided by **EventCallbackAdapter** to obtain the value. For a full list of these helper methods see the API documentation. A few of these helper methods are:

```
// produce the named Attribute's value
std::string getString(const std::string& attributeName);
long getLong(const std::string& attributeName);
double getDouble(const std::string& attributeName);

// produce the event's full value as an AttributeTable
IAttributeTable getAttributeTable();
```

To subscribe the callback to an event stream you should use one of the methods provided in the event helper class **atst::cs::services::Event**.

```
//subscribe the callback to the event stream using the given strategy
void subscribe(const std::string& eventName, IEventCallbackcallback
      callback, const std::string& strategy);

//same as subscribe(eventName, callbck, Event.NORMAL)
void subscribe(const std::string& eventName, EventCallbackcallback
      callback);
```

```
void unsubscribe(const std::string& eventName,  IEventCallbackcallback
      callback);


static final String NORMAL; // the 'normal' subscription strategy
static final String LOSSY; // the 'lossy' subscription strategy
static final String FAST; // the 'fast' subscription strategy
```

5.4.4 C++ Example

The following C++ code fragments demonstrate how one might use the event service to subscribe/publish events to a named event stream.

MyEventListener overrides the callback method of the **EventCallbackAdapter** class. This class demonstrates how one would deduce the data type of the attribute named "value". In practice, the subscriber would expect a specific data type and would only call the conversion method appropriate for that data type.

```
class MyEventListener : public
      atst::cs::services::event::EventCallbackAdapter {

   public:

   void callback(const std::string& eventName) {

      // Identify who sent the event
      std::cout << "event '"+eventName+"' received";

      // Is it a long value?
      try {
         long l = getLong("value");
         std::cout << "long value is: " + l;
         return;
      } catch(...){
         //not a long
      }

      // Is it a double value?
      try {
         double d = getDouble("value");
         std::cout << "double value is: " + dValue;
         return;
      } catch (...) {
         //not a double
      }

      // Finally, is it a string value?
      try {
         std::string sValue = getString("value");
         std::cout << "string value is: " + sValue;
         return;
      } catch (...) {
         //not a string (must not be present)
      }
      std::cout << "Value of '" << eventName << "' not found in event!";
   }
```

```
}
```

The following code fragment demonstrates how an application would subscribe to a named event stream using an instance of **MyEventListener** as the remote callback object.

```
std::::tr1::shared_ptr<MyEventListener> listener =
      std::tr1::shared_ptr<MyEventListener>(new MyEventListener());
Event::subscribe("system.subsystem.device.status", listener);

cout << "Press Q to quit";
char c;
do {
   cin >> c;
} while( (c != 'Q') && (c != 'q'));

std::count << "unsubscribing";
Event::unsubscribe("system.subsystem.device.status", listener);
```

The following code fragment demonstrates how one would publish events of each supported data type to a named event stream.

```
std::string eventName = "system.subsystem.device.status";
std::string aName = "value"

// Post strings
std::cout << "posting strings ...";
for (int i = 0; i < 100; i++) {
   std::stringstream out;
   out << i;
   Event::post(eventName&, aName&, out.str());
}

// Post doubles
std::cout << "posting doubles ...";
for (double d = .1; d < 10.0; d+=.125) {
   Event::post(eventName&, aName&, d);
}

// Post longs
std::cout << "posting longs ...";
for (long l = -100000; l < 100000; l+=5000) {
   Event::post(eventName&, aName&, l);
}
```

6 LOG SERVICE

ATST maintains a permanent record, or *log*, of all system activity. Information is recorded into this log as *messages*. There are two types of log messages:

- **status** – Messages that one would reasonably expect to *always* be logged,
- **debug** – Messages that are only logged during system diagnostics.

ATST Common Services Framework provides a robust, high-performance logging system for recording both types of log messages. All log messages are stored in a relational database and automatically time stamped and the ATST component generating the log message is automatically identified. Log messages may be arbitrarily large but performance is improved if most messages are kept short. The log service provides numerous support operations including the ability of generating a *stack trace* to be logged as an aid in debugging. This stack trace shows the call tree hierarchy from the point of the stack traces back to the root and so can be used to answer the question *"How did I ever wind up here?"*

See TN-0120 for information about tools that can be used to view the log service.

6.1 MESSAGE CATEGORIES

The log service supports grouping messages into broad, cross-component *categories*. The log service defines the following standard categories for use by component developers:

- **DEFAULT** – general log messages
- **STAR** – the wildcard category for overriding other category's levels
- **FLOW** – trace program flow (call/return, branches, and loops)

Additionally developers should feel free to define any additional categories that they would like. User-defined log categories should only use a-z, A-Z, 0-9, _ (underscore), - (hyphen), and : (full colon). Other characters might not be supported by all log service too implementations. For additional details see the reference guide.

Other predefined categories are used to identify messages used within the CSF. These categories, while available for component developers, are intended more for use within CSF itself:

- **EVENT** – trace events received or generated
- **ALARM** – log alarm conditions
- **HEALTH** – log changes to system health
- **CONNECT** – log connections
- **COMMAND** – log commands and responses
- **LIFECYCLE** – log component lifecycle behavior

- **DB** – database internals
- **ARCHIVE** – trace internal operations of the Archive service
- **PROPERTY** – trace internal operations of the property service
- **SCRIPT** – trace internal operations of the script service
- **PARAMSET** – trace internal operations of the parameter set service
- **ACTION** – log action progression

Categories are also useful in controlling actions outside the log system. The section on the Archive Device includes an example where a custom log category is used to control archiving of data.

6.2 STATUS MESSAGES

A status message may belong to one of three classes:

- **note** – messages that are informative but do not indicate a particular problem should belong to this class.
- **warning** – messages indicating a problem that is not yet severe enough to interfere with proper operation of the component belong in this class. If the current debug level is set to non-zero, the message *automatically* append the current method name, source file name, and line number within that file of the call.
- **severe** – messages reporting conditions under which the component is unable to perform normal operation belong in this class. Severe messages include a stack trace if an exception was included.

Status messages are *always* logged.

Note that unconstrained use of status messages does little to enhance the usefulness of the system log and may impose unnecessary strain on resources. Most of the places where status messages are appropriate are within CSF - there is very little need for component developers to add status messages. Places where CSF automatically produces status messages include:

- lifecycle changes
- health changes
- connection changes
- commands and responses
- alarms

Nevertheless, status messaging is available to component developers should specific component functionality suggest their use.

6.3 DEBUG MESSAGES

Debug messages provide information useful when attempting to track down problems and are normally disabled. An operator may enable/disable debug messages at any time on a per category basis. Debug messages are identified by log category, as described above, and by *level*.

6.3.1 Debug Levels

The log service requires debug messages to be identified by a level. In general, the higher the level, the more information is provided for debugging. A current debug level setting determines which debug messages are generated. All messages whose level is less than or equal to the current debug level are produced, so setting the debug level to 2, for example, would cause messages at levels 1 and 2 to be produced. The current debug level is not set by component code, but is managed by the component's lifecycle interface. Every log category maintains its own debug level.

The STAR category is a special purpose debug category that allows the user to open up logging across all categories. When the STAR category is set, all categories will have a minimum level equal to that of the STAR category level. Specific categories can increase their own debug level to a higher value as desired, but if they are set to a lower value than the STAR category debug level, then the STAR category debug level is the one that will be used for subsequent debug messages.

Developers are cautioned that unconstrained debug use can put a strain on system resources and are thus encouraged to set the debug levels of messages judiciously. Putting a level 1 message inside tight loops, for example, is probably not a good idea. Otherwise, determining the appropriate level for a particular debug message is probably more art than science. ATST offers the following recommendations:

- Level 1: at the entry/exit of major code sections (code modules),
- Level 2: at the entry/exit of methods and procedures (unless these are expected to be called within tight loops) and in object constructors (with the same caveat),
- Level 3: at key points within methods and procedures (again, outside of tight loops), and
- Level 4: within tight loops (should be small messages with critical information only).

For example, in the flow category, level 3 messages should be used to identify branches taken in the code and the start of major loops. It is always a bad idea to include stack traces in level 4 debug messages!

6.4 CONVENIENCE METHODS

Additional convenience methods are available that provide features. These are described in the language-specific *helper* sections, below.

6.4.1 Java Helper

The class **atst.cs.services.Log** provides a rich set of static methods for interacting with the logging system:

```
// is debugging enabled in the category?
boolean isEnabled(String category);

// will a debug message be logged at the given category and level?
```

```
boolean isDebuggable(String category, int level);

// produce the category's current debug level
int getDebugLevel(String category);

// log a note in the category
void note(String category, String message);

// log a warning in the category
void warn(String category, String message);

// log a severe condition in the category
void severe(String category, String message);

// log a severe condition in the category with an Exception
void severe(String category, String message, Exception e);

// log a debug message if level <= current debug level
void debug(String category, int level, String message);
```

Since the expectation is that status messages (note, warn, and severe) produced by component developers will likely belong in the default category, the following convenience methods are defined:

```
// log a note in the default category
void note(String message);

// log a warning in the default category
void warn(String message);

// log a severe message in the default category
void severe(String message);

// log a severe message in the default category with an Exception
void severe(String message, Exception e);
```

The next few convenience methods are independent of logging category:

```
// returns a string containing the method name, source file name,
// and line number within that file of the call.
String curLoc();

// returns a string containing a full stack trace from the
// current location in the source code.
String getStackTrace();

// returns, as a string, the full stack trace for the exception ex.
String getStackTrace(Throwable ex)
```

The **getStackTrace** methods can be used to add a stack trace to any log message.

The following are the standard category definitions available to Java Component developers through the log service:

```
String Log.DEFAULT   // default logging category
String Log.STAR      // the wildcard category
```

```
String Log.FLOW      // log flow-control block entry/exit
```

The categories used within CSF are:

```
String Log.Event
String Log.HEALTH
String Log.ALARM
String Log.CONNECT
String Log.COMMAND
String Log.LIFECYCLE
String Log.ACTION
String Log.DB
String Log.ARCHIVE
String Log.PROPERTY
String Log.PARAMSET
String Log.SCRIPT
```

If no category is given, **Log.DEFAULT** is assumed.

6.4.2 Java Example

The following code sample demonstrates the usage of a few capabilities of the logging service.

The entry and exit points of the function are marked with level 2 debug statements in the default category. These debug statements will be logged only if the current debug level for the default category is 2 or greater.

In the arguments parsing, the warning status message is logged if the "**startup.mode**" argument is missing regardless of the current debug level. This warning message also appends an optional stack trace by calling **Log.getStackTrace()**.

The **Log.INIT** category is used for messages relating to hardware initialization. So the debug statement "initializing hardware controller" will be logged in the **Log.INIT** category if the debug level for that category is set to 3 or above. The severe status message will be logged if there is an error during hardware initialization regardless of the debug level.

```
public void doInit() {

   // Entry point
   Log.debug(2, "initializing. In "+Log.curLoc());

   // Parse arguments
   IAttribute mode = Cache.lookup("startup:mode");
   if(mode == null) {
      Log.warn("startup mode unspecified -- stacktrace:" +
         Log.getStackTrace());
   }

   // Hardware initialization
   Log.debug("HW-INIT", 3, "initializing hardware controller");
   if((status = controllerInit(defaultMode)) != OK)
      Log.severe("HW-INIT", "Error initializing hardware controller,
         status = "+status);

   // Exit point
```

```
    Log.debug(2, "initialization complete");
}
```

6.4.3 C++ Helper

The class **atst::cs::services::Log** provides a rich set of static methods for interacting with the logging system:

```
// Is debugging enabled in the category?
bool Log::isEnabled(const string& category);

// Will a debug message be logged at the given category and level?
bool Log::isDebuggable(const string& category, int level);

// Produce the category's current debug level
int Log::getDebugLevel(const string& category);

// Log a note in the category
void Log::note(const string& category, const string& message);

// Log a warning in the category
void Log::warn(const string& category, const string& message);

// Log a severe condition in the category
void Log::severe(const string& category, const string& message);

// Log a severe condition in the category with an exception
void Log::severe(const string& category, const string& message, const
      exception& ex);

// Log a debug message if level <= current debug level
void Log::debug(const string& category, int level, const string& message);
```

Since the expectation is that status messages (**note**, **warn**, and **severe**) produced by component developers will likely belong in the **default** category, the following convenience methods are defined:

```
// Log a note in the default category
void Log::note(const string& message);

// Log a warning in the default category
void Log::warn(const string& message);

// Log a severe message in the default category
void Log::severe(const string& message);

// Log a severe message in the default category with an exception
void Log::severe(const string& message, const exception& ex);
```

The next few methods are independent of logging category:

```
// Returns a string for stack trace from the current location in the
// source code
string Log::getStackTrace();

// Returns a string containing the stack trace for the exception
string Log::getStackTrace(const std::exception& ex);
```

The **getStackTrace** methods can be used to add a stack trace to *any* log message.

The following are the standard category definitions available to C++ Component developers through the log service:

```
const string Log::DEFAULT;       // The default is no category
const string Log::STAR;          // The wildcard category
const string Log::FLOW;          // Program flow
```

The categories used within CSF are:

```
const string Log::ALARM;         // Alarms
const string Log::HEALTH;        // Health
const string Log::CONNECT;       // Connections
const string Log::COMMAND;       // Commands/Responses
const string Log::LIFECYCLE;     // Lifecycle
const string Log::DB;            // Database internals
const string Log::ARCHIVE;       // Archive internals
const string Log::PROPERTY;      // Property internals
const string Log::ACTION;        // Actions
```

If no category is given, **Log::DEFAULT** is assumed.

6.4.4 C++ Example

```
void doInit() {

   // Entry point
   Log::debug(2, "initializing. In "+Log::curLoc());

   // Parse arguments
   pIAttribute mode = Cache::lookup("startup:mode");
   if (!mode){
      Log::warn("startup mode unspecified -- stacktrace:" +
            Log::getStackTrace());
   }

   // Hardware initialization
   Log::debug("HW_INIT", 3, "initializing hardware controller");
   if((status = controllerInit(defaultMode)) != OK){
      Log::severe("HW_INIT", "Error initializing hardware controller,
            status = "+status);
   }

   // Exit point
   Log::debug(2, "initialization complete");
}
```

7 HEALTH SERVICE

The *health service* is used by ATST to maintain the "health" condition of a component. The health may be one of (presented in worsening order):

- **GOOD** – No problems have been detected by the component.
- **ILL** – Problems have been detected, but they do not prevent observing. Data *quality*, however, may be affected. It may also be the case that operation of the component will fail soon if corrective action is not taken.
- **BAD** – Severe problems have been detected. The component is unable to operate correctly. Corrective action is required.
- **UNKNOWN** – The component is not responding. It may or may not be operating. This health value is not set by the component (obviously) but may be set by the health service.

ILL and **BAD** health should be accompanied by a meaningful reason. A reason is optional when the health is **GOOD**.

Components are responsible for recording their health to the health cache. The health service uses the values in the cache to determine a component's health. A health is placed into the cache with a user-defined category. The health service then, in a separate thread, walks the entire cache to determine the worst health and reports that.

The health service automatically posts an event showing changes to the component health and logs a warning on worsening health and a note on improving health. An initial health of Good is logged as a note while all other initial health statuses are logged as a warning.

7.1 JAVA HELPER

The **atst.cs.services.Health** class provides component access to the health service. Component should call the set method or one of the helper setX methods whenever the health of a component may have changed.

```
// get the health of the Component for the given category
public static IHealthStatus get(String category);

// set the health of the Component
public static void set(String category, IHealthStatus status);

// get the worst available health of the Component
public static IHealthStatus getWorst();
// set the health of the Component to good
public static void setGood(String category);

// set the health of the Component to ill with the given reason
public static void setIll(String category, String reason);

// set the health of the Component to bad with the given reason
public static void setBad(String category, String reason);
```

7.1.1 Java Example

The following is an example of a health check function for a subsystem whose health depends on the availability of an internal pool of resources. The health of the subsystem is deemed:

- **BAD** — if there are no more resources available in the pool.
- **ILL** — if there is only one resource available in the pool.
- **GOOD** — if there are two or more available resources.

```
public class PoolWrapper<T> extends Pool<T> {

   public synchronized T getOne() {
      T t = super.getOne();
      if (this.size() == 0) {
         Health.set("pool", HealthStatus.bad("no resources left in
            pool"));
      } else if (this.size() == 1){
         Health.set("pool", HealthStatus.ill("low resources left in
            pool"));
      }
   }

   public synchronized void releaseOne(T t) {
      super.releaseOne(t);
      if (this.size() = 1) {
         Health.setIll("pool", "low resources left in pool");
      } else if (this.size() > 1) {
         Health.setGood("pool");
      }
   }
}
```

The above example wraps some pool resource and sets the component's health based on the resources available. As long as the pool is constructed in the component's namespace, the "pool's health" will be the component's health. Multiple pool wrappers (hopefully with different health categories) could be used in parallel, and the health service would determine which health was the worst and therefore which health should dictate the overall health of the component.

7.2 C++ HELPER

The **atst::cs::services::Health** class provides Component access to the health service.

Component should call the set method or one of the helper setX methods whenever the health of a component may have changed.

```
// set the health of the Component
public static void setHealth(string category, pIHealthStatus status);

// get the health of the Component for the given category
public static pIHealthStatus getHealth(string category);

// get the worst health of the Component
```

```
public static pIHealthStatus getWorstHealth();

// set the health of the Component to good
public static void setGood(string category);

// set the health of the Component to ill with the given reason
public static void setIll(string category, string reason);

// set the health of the Component to bad with the given reason
public static void setBad(string category, string reason);
```

7.2.1 C++ Example

```
template<typename T> class PoolWrapper : public Pool<T> {
   public:
      T getOne() {
         T t = PoolgetOne();
         if (this.size() == 0) {
            Health::set("pool", HealthStatus::bad("no resources left in
                pool"));
         } else if (this.size() == 1){
            Health::set("pool", HealthStatus::ill("low resources left in
               pool"));
         }
      }

      void releaseOne(T t) {
         Pool::releaseOne(t);
         if (this.size() = 1) {
            Health::setIll("pool", "low resources left in pool");
         } else if (this.size() > 1) {
               Health::setGood("pool");
         }
      }
}
```

8 PROPERTY SERVICE

The *Property Service* maintains *metadata* about attributes in a *persistent store*. This metadata is used by the technical architecture to determine if an attribute is valid and should be passed down the functional architecture. This means that functional code does not have to check to see if an attribute is properly typed, and within the necessary limits. This metadata (the *properties* of the attribute) consists of:

- **type** - the type of the attribute's value(s). This is an *ATST type*, not an implementation language type. For example, a simple position is a **Position** while a 2-D position is a **XYPosition**. At the current time the following types exist (more types will be added):
    - **"string"** – an arbitrary-length string,
    - **"integer"** – a signed, integral value of up to 64 bits,
    - **"real"** – a signed, floating-point value of 64 bits,
    - **"boolean"** - a simple Boolean value,
- **vector** – a Boolean flag that is true if the attribute is a *vector* of type. Note that some types of attributes are themselves vector-valued. This flag would denote a *vector of vectors* in such cases.
- **permission** – similar to Unix file permissions an attribute can be readable, writeable, or executable. A readable attribute can be queried via a get. A writeable attribute can be passed via a set. An executable attribute can be passed via submit. An attempt to pass an attribute via get or set without the necessary permission will cause the attribute to be removed from the table, and a warning in the log. The partial table will then be passed to the functional code. An attempt to pass an attribute via submit without the necessary permission, will cause the entire configuration to be rejected as bad parameter without the function code every being called.
- **description** – a description of the attribute.
- **values** – the Attribute's value(s). Applications may record the current value of an Attribute to provide *persistency*.
- **defaults** – the default value(s) for the attribute, if any.
- **limits** – bounds of legal values, if any. The values here depend upon the **`type`**:
    - **string** limits, if they exist, denote exact values that are legal strings. In essence, this means that the property denotes an *enumeration*. If there are no limits, then any string is considered legal.
    - **integer** limits, if they exist, denote a (low, high)-inclusive range. If only a single limit is given, it is the lowest value allowed.
    - **real** limits, if they exist, denote a (low ,high)-inclusive range. If only a single limit is given, it is the lowest value allowed.
    - **boolean** limits don't exist since it is a self-limiting type.

- **changeDeltas** – changes below these deltas, if any, are not monitored. Not all types have change deltas, but those that do typically have two values: **lowDelta** and **highDelta**. Change deltas are only meaningful on numeric-valued data.

- **mask** – The mask allows values within a vector type attribute to be selectively passed, blocked, or required. The mask itself is a vector, and its individual elements may have one of four values.

    - **ALLOW** -or- **A** – Elements that are allowed will be passed by the technical architecture to the functional architecture unmodified (assuming the element also meets all other metadata constraints). Null values will be passed along under this mask.

    - **BLOCK** -or- **B**– Elements that are blocked will be selectively removed from the vector. In the case of java this means that a null value will be present. In the case of c++ the empty string will be present.

    - **REQUIRE** -or- **R**– Elements that are required are treated just like allowed elements if they are present (non-Null). If a required element is not present then the entire attribute will be treated is invalid and the technical architecture will not pass it to the functional architecture.

    - **IGNORE_AFTER** -or- *– This wildcard mask is only valid as the last element in the mask vector. It indicates that zero or more elements may follow in the data vector and they will all be treated as if masked with ALLOW.

Every property has a *name* and is associated uniquely with a *source* (typically a component or controller). Property names are *fully-qualified* – they consist of a *simple name* prepended by the source name. A simple name is defined as a name with no embedded periods ("**.**"). While the property service allows referencing properties using the simple name, the service automatically prepends the source name to fully-qualify the property name. The fully-qualified name matches the name of the attribute that has these properties associated with it.

8.1 PROPERTIES VERSUS CONSTANTS

The Property Service provides a mechanism that allows components access to metadata on *application-specific* attributes. This can be distinguished from other types of information such as *manifest constants* that are immutable across all ATST applications. Examples of such constants include the information that uniquely describes the precise location (*latitude*, *longitude*, and *elevation*) of ATST. Such manifest constants are provided by the Constant Service. While the implementation internals of the Constant Service are built on top of the property service implementation, the two services are distinct at the component level.

8.2 COMPONENT ACCESS TO ATTRIBUTE METADATA

The **`Property`** access helper provides Component developers with the following methods:

- **exists(attributeName)** – true if a property set for this attribute exists.

- **getType(attributeName)** – produce the type (as a string) of the named attribute.

- **isVector(attributename)** – returns **true** if this attribute is a vector of the indicated **type.**

- **isReadonly(attributeName)** – returns **true** if this attribute is read-only.
- **getDescription(attributeName)** – produce the description of the attribute
- **saveAttribute(attribute)** – record the value of attribute into the persistent store.
- **getAttribute(attributeName)** – produce an attribute for the named attribute, with the saved value restored. If no values have been saved, then the *default* values are used. Note that, as an attribute, the value is represented as an array of strings.
- **getDefault(attributeName)** – produce an attribute for the named attribute, with all values set to their defaults.
- **getValues(attributeName)** – produce any saved values.
- **getDefaults(attributeName)** – produce the default values.
- **getLimits(attributeName)** – produce the limits values.
- **getDeltas(attributeName)** – produce the change deltas.

A **null** is returned if the requested metadata item does not exist for the named attribute.

There are a series of *convenience* methods for performing various tests against the properties:

```
// is simpleValue within limits?
inRange(attributeName, simpleValue);

// is simpleValue (a string) within limits?
inRangeString(attributeName, simpleValue);

// are all simpleValues (strings) within limits?
inRangeStrings(attributeName, simpleValues);
```

Several additional convenience methods are also defined to handle other common cases:

```
// set values as the saved values of the attribute's property
setValues(attributeName, values);

// set value as the first saved value of the attribute's property
setValue(attributeName, value);

// produce the first saved value of the attribute's saved value
getValue(attributeName);
```

Note that there is *no support* for a Component altering any of the metadata except for the saved values. To change any of the other metadata a user should use one of the property tools described in TN-0120.

A special method is provided for the few cases where a Component needs access to the properties of a *different* component:

```
// returns a table holding all the properties for the named Component
getRemoteProperties(appName);
```

This situation is expected to be limited to components that *manage* other components, or components that control transient physical objects. For example: a camera that can be plugged into different computers, and managed by different, components might have its details stored by camera id/name, so that only the camera name property of the component would need to be changed if the camera was switched out.

The service access helpers for each language may provide additional convenience methods, see below for details.

Instead of needing to explicitly define all properties for an application, one or more generic property sets may be. If a property "&inherits" is present in an application's property table, the property service will attempt to 'inherit' the properties found in the tables named in the "&inherits" property. Tables are inherited in a right to left, depth first manner with the property table doing the inheriting considered the furthest right. If a property is found to be present in more than one table, the property found in the right-most table is used. This can be used to define a set of generic properties for, say, a multi-threaded application vs. a single-threaded application and then reuse those two sets in many different applications.

When a property is inherited, the original fully-qualified-prefix (as determined by the name of the entity 'owning' the property) is replaced with the inheriting component's name. So if the component atst.tcs were to inherit properties from a default table such as:

```
Owner       Propery Name   type/values/...
default     default.foo    ...
default     default.bar    ...
```

The atst.tcs component would appear to have defined the properties:

```
Owner       Property Name  type/values/...
atst.tcs    atst.tcs.foo   ...
atst.tcs    atst.tcs.bar   ...
```

8.3 JAVA PROPERTY SERVICE HELPER

In Java, properties are kept in their *native format* when held in memory (e.g., .in Java *real*-valued properties are kept as **Double** values, *integer*-valued properties as **Long** values, etc.).

The implementation of the access helper is straightforward and provides Java implementations of the above methods:

```
// true if a property set for this attribute exists
boolean exists(String attributeName);

// produce the attribute's type
String getType(String attributeName);

// true if attribute is a vector of the indicated type
boolean isVector(String attributeName);

// true if attribute is readable
boolean isReadable(String attributeName);

// true if attribute is writeable
boolean isWriteable(String attributeName);

// true if attribute is executable
boolean isExecutable(String attributeName);
```

```
// produce the attribute's description
String getDescription(String attributeName);

// save the attribute's value into the persistent store
void saveAttribute(atst.cs.interfaces.IAttribute attribute);

// produce an Attribute for the named attribute using the saved values
atst.cs.interfaces.IAttribute getAttribute(String attributeName);

// produce an Attribute for named attribute w/all values set to defaults
atst.cs.interfaces.IAttribute getDefault(String attributeName);

// produce the array of saved values
Object[] getValues(String attributeName);

// produce the array of default values
Object[] getDefaults(String attributeName);

// produce the array of limit values.
Object[] getLimits(String attributeName);

// produce the array of change deltas
Object[] getDeltas(String attributeName);

// produce the properties for the named Component
atst.cs.interfaces.IPropertyTable getRemoteProperties(String appName);
```

Note that casts are required to properly use **getValues**, **getLimits**, **getDeltas**, and **getDefaults**, and that the user is responsible for understanding the meaning of each array element for each.

Convenience methods are provided for performing tests against the property:

```
// is value within the limits of the named attribute?
boolean inRange(String attributeName, String value)

// is value within the limits of the named attribute?
boolean inRange(String attributeName, Long value)

// is value within the limits of the named attribute?
boolean inRange(String attributeName, Double value)

// is sValue (the String form for a value of the property's type)
// within the limits of the named attribute?
boolean inRangeString(String attributeName, String sValue)

// are sValues within the limits of the named attribute?
// Returns true only if all are within limits.
boolean inRangeStrings(String attributeName, String[] sValues)
```

This convenience method can be used to set the entire array of saved values for the property of the named attribute:

```
void setValues(String attributeName, String[] values);
```

The Java service access helper also provides the following convenience methods for dealing with the metadata in a vector property:

```
// produce the current values as an array of Strings.
String[] getStringArray(String attributeName);

// produce the current values as an array of Longs, if the
// property is a vector of integers.
Long[] getLongArray(String attributeName)

// produce the current values as an array of Doubles, if the
// property is a vector of reals.
Double[] getDoubleArray(String attributeName)

// produce the current values as an array of Booleans, if the
// property is a vector of booleans.
Boolean[] getBooleanArray(String attributeName)
```

You can also obtain limits, deltas, etc. as arrays of Strings:

```
String[] getStringDefaults(String attributeName);
String[] getStringLimits(String attributeName);
String[] getStringDeltas(String attributeName);
```

Some additional convenience methods are included to handle the common case where the value of the attribute is a single string value:

```
// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(String attributeName, String value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(String attributeName, Long value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(String attributeName, Double value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(String attributeName, Boolean value);

// return the saved value of the property. Note that the result
// must be cast to the appropriate type for the named property.
Object getValue(String attributeName);

// return the saved value of the property as a String.
String getString(String attributeName);

// return the saved value of the property as a Boolean.
Boolean getBoolean(String attributeName);

// return the saved value of the property as a Long.
Long getLong(String attributeName);
```

```
// return the saved value of the property as a Double.
Double getDouble(String attributeName);
```

In all of these convenience methods, the value is the *first* value held in the property, other values, if any, are lost.

A special method is provided for the few cases where a Component needs access to the properties of a *different* component:

```
// returns a table holding all the properties for the named Component
IPropertyTable getRemoteProperties(String appName);
```

8.3.1 Java Example

The following code sample shows how one might obtain the limits for a real-valued attribute **anAttribute**. These limits could be used, for example, to verify values for this attribute in a user interface.

```
Double[] limits = (Double [])Property.getLimits("anAttribute");
```

If the variable **dValue** contains a **Double**, the following code tests **dValue** against those same limits:

```
if (Property.inRange("anAttribute", dValue)) {
        // code to perform if value is within limits
}
```

Similarly, an attribute containing the default values for **anAttribute** can be obtained with:

```
IAttribute anAttribute = Property.getDefault("anAttribute");
```

8.4 C++ HELPER

In contrast to the Java implementation in C++, properties are kept in their *stringified format* when held in memory (e.g., in the *real*-valued property 1.0 would be stored as the string "1.0").

The implementation of the access helpers is straightforward and provides native implementations of the above methods:

```
// true if a property set for this attribute exists
bool exists(const std::string& propertyName);

// produce the attribute's type
std::string getType(const std::string& propertyName);

// true if attribute is a vector of the indicated type
bool isVector(const std::string& propertyName);

// true if attribute is readable
bool isReadable(const std::string& propertyName);

// true if attribute is writable
bool isWritable(const std::string& propertyName);

// true if attribute is executable
```

```
bool isExceutable(const std::string& propertyName);

// produce the attribute's description
String getDescription(const std::string& propertyName);

// save the attribute's value into the persistent store
void saveAttribute(IAttribute attribute);

// produce an Attribute for the named attribute using the saved values
std::tr1::shared_ptr<IAttribute> getAttribute(const std::string&
      propertyName);

// produce an Attribute for named attribute w/all values set to defaults
std::tr1::shared_ptr<IAttribute> getDefault(const std::string&
      propertyName);

// produce the array of saved values
std::vector<{TYPE}> get{TYPE}Values(const std::string& propertyName);
// where {TYPE} is one of integer, real, boolean, string

// produce the array of default values
std::vector<{TYPE}> get{TYPE}Defaults(const std::string& propertyName);

// produce the array of limit values.
std::vector<{TYPE}> get{TYPE}Limits(const std::string& propertyName);

// produce the array of change deltas
std::vector<{TYPE}> get{TYPE}Deltas(const std::string& propertyName);

// produce the properties for the named Component
atst::cs::interfaces::IPropertyTable getRemoteProperties(const
      std::string& appName);
```

Note that casts are required to properly use **getValues**, **getLimits**, **getDeltas**, and **getDefaults**, and that the user is responsible for understanding the meaning of each array element for each.

Convenience methods are provided for performing tests against the property:

```
// is value within the limits of the named attribute?
bool inRange(const std::string& attributeName, const std::string& value)

// is value within the limits of the named attribute?
bool inRange(const std::string& attributeName, Long value)

// is value within the limits of the named attribute?
bool inRange(const std::string& attributeName, Double value)

// is sValue (the String form for a value of the property's type)
// within the limits of the named attribute?
bool inRangeString(const std::string& attributeName, String sValue)

// are sValues within the limits of the named attribute?
// Returns true only if all are within limits.
bool inRangeStrings(const std::string& attributeName,
      std::vecotr<std::string> sValues)
```

This convenience method can be used to set the entire array of saved values for the property of the named attribute:

```
void setValues(const std::string attributeName&, std::vector<std::string>
      values);
```

The C++ service access helper also provides the following convenience methods for dealing with the metadata in a vector property:

```
// produce the current values as an array of Strings.
std::vector<std::string> getStringArray(const std::string& attributeName);

// produce the current values as an array of Longs, if the
// property is a vector of integers.
std::vector<int64_t> getLongArray(const std::string& attributeName)
)

// produce the current values as an array of Doubles, if the
// property is a vector of reals.
std::vector<double> getDoubleArray(const std::string& attributeName)

// produce the current values as an array of Booleans, if the
// property is a vector of booleans.
std::vector<bool> getBooleanArray(const std::string& attributeName)
```

You can also obtain limits, deltas, etc. as arrays of Strings:

```
std::vector<std::string> getStringDefaults(const std::string&
      attributeName);
std::vector<std::string> getStringLimits(cosnt std::string&
      attributeName);
std::vector<std::string> getStringDeltas(const std::string&
      attributeName);
```

Some additional convenience methods are included to handle the common case where the value of the attribute is a single string value:

```
// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(const std::string& attributeName, const std::string& value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(const std::string& attributeName, int64_t value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(const std::string& attributeName, double value);

// set the saved value of the property for the named attribute
// to the indicated value.
void setValue(const std::string& attributeName, bool value);

// return the saved value of the property as a String.
std::string getString(const std::string& attributeName);
```

```
// return the saved value of the property as a Boolean.
bool getBoolean(const std::string& attributeName);

// return the saved value of the property as a Long.
int64_t getLong(const std::string& attributeName);

// return the saved value of the property as a Double.
double getDouble(const std::string& attributeName);
```

In all of these convenience methods, the value is the *first* value held in the property, other values, if any, are lost.

A special method is provided for the few cases where a Component needs access to the properties of a *different* component:

```
// returns a table holding all the properties for the named Component
pIPropertyTable getRemoteProperties(const std::string& appName);
```

8.4.1 C++ Example

The following code sample shows how one might obtain the limits for a real-valued attribute **anAttribute**. These limits could be used, for example, to verify values for this attribute in a user interface.

```
std::vector<std::string> limits =
      Property::getStringLimits("anAttribute");
```

If the variable **dValue** contains a **double**, the following code tests **dValue** against those same limits:

```
if (Property::inRange("anAttribute", dValue)) {
      // code to perform if value is within limits
}
```

Similarly, an attribute containing the default values for **anAttribute** can be obtained with:

```
Shared_ptr<IAttribute> anAttribute = Property::getDefault("anAttribute");
```

9 ALARM SERVICE

The *alarm service* provides a convenient facility for raising alarms. Components raise alarms in response to exception conditions that require immediate operator intervention.

9.1 JAVA HELPER

The **atst.cs.services.Alarm** class contains a static method to support raising alarms:

```
// raises an alarm
void raise(String alarmType, String description);
```

There is one method provided for monitoring alarm events, intended for use with the alarm display system:

```
// respond to a raised alarm
void monitor(atst.cs.util.IMesgCallback callback);
```

The callback is executed whenever a raised alarm is received.

9.2 C++ HELPER

The **atst::cs::services::Alarm** class contains a static method to support raising alarms:

```
// raises an alarm
void raise(const std::string &alarmType, const std::string &description
      );
```

There is one method provided for monitoring alarm events, intended for use with the alarm display system:

```
// respond to a raised alarm
void monitor(std::shared_ptr<atst::cs::util::IMesgCallback> callback);
```

The callback is executed whenever a raised alarm is received.

10 MINOR SERVICES

The ATST common service *tools* (aka "*minor services*") provide generally-useful support for component developers. In many cases these tools expose functionality used internally within CSF.

10.1 ARCHIVE SERVICE

The *archive service* provides high-performance archiving of **Attributes** (name/value pairs). A timestamp and the name of the source component are automatically recorded with each **Attribute**. The intent is to provide a means of saving bursts of engineering data for later analysis. Normally, developers should wrap calls to the archive service within tests to control the production of archived data. For example, developers could do this (using pseudo-code that only happens to look like Java):

```
Log.note("TCS_ARCHIVING","Archiving target position stream");
....
if (isEnabled("TCS_ARCHIVING")) {
         Archive.archive(mcsPosition);
}
....
Log.note("TCS_ARCHIVING","Done archiving target position stream");
```

Note that this example uses a log service *category* to control access to the archiver. The log service *debug level* can also be used to refine this control.

At the current time, there is no programmatic support in the archive service for retrieving values that have been archived. The archived data is maintained in a relational database and **SQL** commands may be used to retrieve values from the archive. Applications that support the analysis of archived data are expected to be added to the CSF at some point in the future.

10.1.1 Java Helper

The **atst.cs.services.Archive** class contains some static methods to access the archive service:

```
// archives attribute
void archive(IAttribute attribute)

// archive a name/value pair
void archive(String name, String value[])
void archive(String name, String value)
void archive(String name, double value)
void archive(String name, double[] value)
void archive(String name, long value)
void archive(String name, long[] value)
```

In the second case, it should be noted that **value** is an array of strings. This name/value pair is used to construct an **Attribute** for archiving, so there is no performance advantage in using the second form – it is merely a convenience method. This holds true for the last two methods also.

10.1.2 C++ Helper

```
// archives attribute
void archive(std::tr1::shared_ptr<IAttribute> attribute)

// archive a name/value pair
```

```
void archive(const std::string& name, std::vector<std::string> value)
void archive(td::string&name, std::string& value)
void archive(td::string&name, double value)
void archive(const std::string& name, std::vector<double> value)
void archive(td::string&name, int64_t value)
void archive(const std::string& name, std::vector<int64_t> value)
```

10.2 CONSTANT SERVICE

The *Constant Service* maintains ATST *manifest constants*: values that are immutable and uniform across all ATST applications. Examples of such manifest constants include the precise location (*latitude*, *longitude*, and *elevation*) of ATST.

At the current time, the following manifest constants exist:

- **latitude** – position of ATST
- **longitude** – position of ATST
- **elevation** – position of ATST

Systems may add their own manifest constants to this service, but must do so manually.

10.2.1 Component Access to Manifest Constants

The **`Constant`** access helper provides Component developers with the following methods:

```
// produce the value (as a string) of the named constant.
getValue(constantName);

// produce the description of the named constant.
getDescription(constantName);
```

In addition, the following *convenience methods* are defined:

```
// produce the constant value as a floating point number.
getDouble(constantName);

// produce the constant value as an integer.
getLong(constantName);
```

A **null** is returned if the requested metadata item does not exist for the named attribute or if an impossible conversion is required.

10.2.2 Java Property Service Helper

The implementation of the access helper is straightforward and provides Java implementations of the above methods:

```
// produce the constant's description.
String getDescription(String constantName) ;

// produce the constant's value, returning null if that isn't possible.
String getValue(String constantName);
```

```
Double getDouble(String constantName);
Long getLong(String constantName);
```

10.2.2.1 Java Example

The following code sample shows how one might obtain the latitude for ATST:

```
String latitude = Constant.getValue("latitude");
```

10.2.3 C++ Helper

```
// produce the constant's description, throwing an exception
// if he constant wasn't found
std::string getDescription(const std::string& constantName) ;

// produce the constant's value throwing an exception
// if doesn't exist or its type doesn't match.
std::string getValue(const std::string&constantName);
double getDouble(const std::string&constantName);
long getLong(const std::string&constantName);
```

10.2.3.1 C++ Example

The following code sample shows how one might obtain the longitude for ATST:

```
std::string longitude = Constant::getValue("longitude");
```

10.3 USER INTERFACES SUPPORT

The ATST Common Services Framework provides basic support for implementing user interfaces.

CSF provides a tool (JES) for constructing graphical user interfaces. JES provides a set of widgets that are customized for use with CSF-based application. See the JES User's Manual (TN-0089) for details.

10.4 MISCELLANEOUS SERVICES

This section covers a variety of minor services provided by Common Services Framework. In most cases they are simply introduced here. Most are either too simple or too obscure to warrant much discussion. Others are language specific. Details of their use can be found in the source code documentation.

10.4.1 Thread Support

The ATST Common Services Framework provides a few tools to support threaded programming. Specifically, there is a simple implementation of a *thread pool* used inside the common services that is exposed for use by Component developers. As an example, thread support is utilized by the Controller class to provide the action threads. Details on the use of the ATST thread pools can be found in the Java API documentation for the **atst.cs.util.threads** package or in the C++ API documentation for the **atst::cs::util::threads** package.

10.4.2 Generic Pools

Besides thread pools, other types of *pools* (collections of isomorphic resources) are possible. The ATST Common Services Framework provides a generic mechanism for managing such pools. Pools may be

fixed in size or automatically **growable** as pool resources are depleted and can by dynamically switched back and forth between **fixed** and **growable**. The pool keeps track of both *active* and *unallocated* resources.

Details can be found in the Java API documentation for the **atst.cs.util.Pool** class. Details can be found in the C++ API documentation for the **atst::cs::util::Pool** class.

10.4.3 ID Service

The *ID service* is used internal to the ATST common services but is exposed in case component developers need a similar functionality. This service provides identification strings that are *guaranteed* to be unique across *all* of ATST. The service is high-performance. The ID strings contain a number that is monotonically increasing but not necessarily sequential.

10.4.3.1 Java helper

The class **atst.cs.services.IdDB** provides access to the ID service. A single static method is available:

```
// returns a unique id beginning with prefix.
String getId(String prefix);
```

The choice of a **prefix** is a convenience choice – the ID will be unique regardless of the prefix.

10.4.3.2 C++ Helper

The class **atst::cs::services::IdDB** provides access to the ID service. A single static method is available:

```
// returns a unique id beginning with prefix.
std::string getId(const std::string& prefix);
```

10.4.4 Cache

The cache provides a single location to find saved/default/loaded values. Because attribute tables can be passed to a component before it has been initialized, the information must be stored somewhere until the component is ready to use it. The mechanism for storing this information is the Cache. The technical architecture places any attribute that is passed to a component by a **set()** command into the cache. The information then waits there for latter accessing. The cache provides a convenient fallback approach to retrieving the information. When the cache is queried, it first looks to see if an attribute has been passed to it. If no attribute was explicitly passed to the cache it then checks the property service to see if a property exists. If the property service has an entry the cache will first try to obtain the saved value. If there is no saved value the cache will obtain the default value. If there is no default value the Cache will log a debug message with the name of the missing property. There is a unique Cache for each component. All Log message from Cache are of the category "CACHE".

10.4.4.1 Java Helper

The class **atst.cs.util.Cache** provides access to the Cache service. The primary methods of interest are:

```
static IAttribute lookup(String name);
static IAttributeTable lookupAll(IAttributeTable attributeTable);
static IAttributeTable lookupAll(String… names);
static IAttribute lookupWithDefault(IAttribute defValue);
```

```
static void contains(String name);
static void store(IAttribute attribute);
static void remove(String name);
static IAttributeTable getCopy();
```

For details on these and other methods see the API documentation.

10.4.4.2 C++ Helper

The class **atst::cs::util::Cache** provides access to the Cache service. The primary methods of interest are:

```
static pIAttribute lookup(std::string& name);
static pIAttributeTable lookup(pIAttributeTable table);
static pIAttributeTable lookup(std::vector<std::string> names);
static pIAttribute lookupWithDefault(pIAttribute defValue);
static void contains(std::string& name);
static void store(pIAttribute attribute);
static void storeAll(pIAttributeTable table);
static void remove(std::string& name);
static pIAttributeTable getCopy();
```

For details on these and other methods see the API documentation.

10.4.5 Time Service

The Time service provides components with a way of querying the current TAI time. The accuracy of the result will depend on the service tool being used.

10.4.5.1 Java Helper

The class **atst.cs.services.Time** provides access to the Time service. The only method is:

```
static IAtstDate getTAITime();
```

10.4.5.2 C++ Helper

The class **atst::cs::services::Time** provides access to the Time service. The only method is:

```
static pAtstDate getTAITime();
```

10.4.6 Date Service

Dates and times are heavily used in ATST, as is the case with most observatories. The **GregorianCalendar** class provided by Java has the unfortunate property of producing *massive* serialized forms. The ATST **atst.cs.util.AtstDate** class provides an alternative to **GregorianCalendar** that produces a much smaller serialized form. Details can be found in the Java API documentation for the **atst.cs.util.AtstDate** class. C++ also provides an equivalent date class found at **atst::cs::util::AtstDate**. Its details can be found in the C++ API documentation.

10.5 SCRIPTING SUPPORT

The Common Services Framework provides support for directly executing scripts from within components. At the current time, two languages are supported for scripts: Java and Python. Components and Controllers may interact with scripts by setting and getting the values of script variables. Additionally, all scripts have full access to the Common Services Framework services and tools.

Scripting support is available by requesting a ***script executor*** that acts as the interface between the component and the script. All script executors provide the same common interface for use by the enclosing component. A script executor accepts an entire script once and executes it. An alternative is a ***script interpreter***, which is capable of executing a script one fragment at a time.

Scripting is available only in Java. There are no plans to implement C++ scripting support.

10.5.1 Script Executors

Java components get a script executor via the **atst.cs.util.scripting.ExecutorFactory** class using the static method:

```
public static IExecutor getExecutor(String executorName);
```

While it is possible to provide executors for different scripting languages, there is currently only a single choice for requesting a Python script executor: **Jython**

All script executors implement the **atst.cs.util.scripting.IExecutor** interface, which defines the following methods:

```
// pass the script contained in scriptString to the executor. The
// script is given the name in scriptName, which is used to identify
// that script as the source of events, log messages, alarms, etc.,
// that may be generated  during the execution of the script.
public void setScript(String scriptName, String scriptString);

// set the value of the script variable named varName to value.
public void setParam(String varName, Object value);

// produce the current value of  the script variable varName.
public Object getParam(String varName);

// parse the script looking for syntax errors. Some executors may
// also compile the script at this time for faster execution.  This
// method returns true if the script is syntactically error free.
// (Some executors are unable to preparse scripts.  In these cases,
// parse() always returns true.  At this time, only the GroovyExecutor
// can accurately parse scripts without running them.)
public boolean parse();

// execute the script.  Automatically parses the script if parse()
// as not been called, but does not report any syntax errors. Syntax
// errors are logged, however.
public void run();
```

10.5.2 Script Interpreters

Java components get an interpreter via the **atst.cs.util.scripting.InterpreterFactory** class using the static method:

```
public static IInterpreter getInterpreter(String interpreterName);
```

While it is possible to provide interpreters for different scripting languages, there is currently only a single choice for requesting a Python script executor: Jython

All script interpreters implement the **atst.cs.util.scripting.IInterpreter** interface, which defines the following methods:

```
// execute the code fragment contained in scriptFragment.  This method
// may be called repeatedly with successive fragments of the script.
// Each fragment must be syntactically complete (e.g. an entire while
// loop, for example).
public boolean evaluate(String scriptFragment);

// set the value of the script variable named varName to value.
public void setParam(String varName, Object value);

// produce the current value of the script variable varName.
public Object getParam(String varName);

// Release any resources held by the interpreter.  This method should
// always be called when finished with the interpreter.
public void endShell();
```

10.5.3 Common Services Framework support for scripts

Scripts automatically have full access to the following CSF Java packages (other CSF packages may be accessed explicitly):

```
atst.cs.data
atst.cs.interfaces
atst.cs.services
atst.cs.services.app
atst.cs.util
atst.cs.util.gui
atst.cs.util.threads
```

Jython scripts are written in standard Python code fragments. For example, the following simple script can be run with the *jython* executor or interpreter:

```
for i in range(limit.getInteger(),-1,-1):
    print '\t'+`i`
    Event.post('eventTest', i)
    Misc.pause(delay)
x = 2 + limit.getInteger()
```

There are several interesting points to note about the above script:

- The script makes two CSF service calls. Events are posted using the CSF event service, and delays are specified using the CSF **Misc** utilities.

- The variables **limit**, **delay**, and **x** have not been declared. Variables used in scripts do not need to be explicitly declared.

- The variables **limit** and **delay** are used without having values assigned to them within the script. Variable values may be set by the enclosing component prior to running the script.

- Similarly, the value of **x** is set without being used. The values of variables computed in the script may be queried by the enclosing component.

10.5.4 Java examples

The following example `main` method allows the user to specify the executor and the values of the countdown **limit** and **delay** between counts. Once the executor has been chosen, the method selects the appropriate version of the above example scripts, sets the **limit** and **delay** parameters and then runs the script. When the script completes execution, the method retrieves the value of **x** from the script executor and displays it. (This example is taken from the **atst.cs.util.scripting.ExecutorFactory.Test** class.)

```
public static void main(String args[]) {
         Misc.parseArgs(args);
   if (Misc.getArg("help", false)) {
       helpMesg();
       System.exit(0);
   }
   String executorName = Misc.getArg("shell", "Jython");
   Integer limit = Misc.getArg("limit", 10);
   Integer delay = Misc.getArg("delay", 1000);

   StandAlone.startServices(executorName+" script demo",
                                "atst.cs.ice.IceToolBoxLoader");
   System.out.println("Testing scripts with "+executorName);



   String pScript = "";
   pScript += "for i in range(limit.getInteger(),-1,-1):\n";
   pScript += "\tprint '\t'+`i`\n";
   pScript += "\tEvent.post('eventTest', i)\n";
   pScript += "\tMisc.pause(delay)\n";
   pScript += "x = 2 + limit.getInteger()\n";

   String script = null;
   if ("jython".equals(executorName.toLowerCase())) script = pScript;

   IExecutor executor = ExecutorFactory.getExecutor(executorName);
   Object result = null;
   if (null != executor) {
       executor.setScript("Countdown timer", script);
       if (executor.parse()) {
           executor.setParam("limit", new Attribute("limit",limit));
           executor.setParam("delay", delay);
           System.out.println("Running:\n"+script+"\n\n");
           executor.run();
           result = executor.getParam("x");
           System.out.println("on return from script, x = "+result);
       }
  }
  StandAlone.stopServices();
}
```

An example runs showing the use of Jython follows:

```
->ajava atst.cs.util.scripting.ExecutorFactory\$Test --limit=5 \
                                          --delay=100 --shell=jython
Testing scripts with jython
Running:
```

```
for i in range(limit.getInteger(),-1,-1):
        print ' '+`i`
        Event.post('eventTest', i)
        Misc.pause(delay)
x = 2 + limit.getInteger()

        5
        4
        3
        2
        1
        0
on return from script, x = 7
->
```

A utility, **RunScript**, is provided by CSF to allow for the testing and execution of arbitrary scripts. For example:

```
->RunScript –noprompt
limit = Attribute('limit', 5)
delay = 100
for i in rage(limit.getInteger(),-1,-1):
    print ' '+`i`
    Event.post('eventTest','value',i)
    Misc.pause(delay)
      5
   4
   3
   2
   1
   0
->
```

More details on **RunScript** can be found in TN-0120

As a more sophisticated example, the following Jython script runs in conjunction with the *ATST TCS demonstration*. The script moves the target position to the four 'corners' of the Sun (North, West, South, East). Once each position is reached, the script pops up a confirmation dialog box and waits for user confirmation before continuing to the next spot. After visiting all four corners, the script returns the target to the original position on the Sun. Note the use of callbacks to handle the confirmations.

```
# Move to the four 'corners' of the sun
from atst.cs.controller import ActionCallback

locs = [ "North",   "West",    "South",    "East"     ]
pos  = [ [0.0,1.0], [1.0,0.0], [0.0,-1.0], [-1.0, 0.0] ]

class CB (ActionCallback):
    msg = ""
    def __init__(self, newMsg):
        ActionCallback.__init__(self)
        self.msg = newMsg
    def doDone(self, config):
        MiscControl.confirmContinue(self.msg+" of Sun.\nReady to
      proceed?")
```

```
def doMove(cmp, msg, target):
    cb = CB(msg)
    cmp.submit(target, cb)
    cb.waitForDone(250)

def setTarget(targetSys, x, y):
    table = AttributeTable()
    table.insert(Attribute("coord", "helioprojective"))
    table.insert(Attribute("x", x))
    table.insert(Attribute("y", y))
    config = Configuration("tcs input")
    config.merge(table.requalify(targetSys))
    return config

TCS = "atst.tcs"
tcs = App.connect(TCS)
tcsInput = App.connect("atst.tcs.input")

curLoc = tcsInput.get(setTarget(TCS, 0.0, 0.0))
for i in range(0,len(locs)):
    doMove(tcs, locs[i], setTarget(TCS, pos[i][0], pos[i][1]))
curLoc.insert(Attribute("send"))
tcsInput.set(curLoc)

App.disconnect("atst.cs.input")
App.disconnect(TCS)
```

10.6 SCRIPT DATABASE

CSF provides a database for storing scripts. Scripts are stored in database are plain text. They are can be accessed by specifying one or more of the given database parameters:

- Category – the system that 'owns' the script. The convention is to use the system tag (e.g. atst.tcs, atst.ocs, atst.ics.vbi, etc.) as the category.

- Name – the identifying name for the script within the category. In the case of instrument scripts, this is typically the observation mode (e.g. observe, gain, setup, etc.). For OCS scripts, this is typically the operation mode.

- ID – a distinguishing tag for different scripts within a given category and name. For example, the scripts within the atst.ocs category and named polcal for the PolCal operation mode may have ids such as generic, tableA, tableB, etc.)

- Version – unique key for any script. Regardless of category, name, and id, all scripts are automatically tagged with a unique version number. The version number is updated whenever a script is inserted or updated in the script database.

For a complete list of script storage/accessor methods see the source or api documents for atst.cs.services.ScriptDB (Java) or atst::cs::services::ScriptDB (C++).

10.7 PARAMETER SET DATABASE

CSF provides a database for storing parameter sets. Fundamentally a parameter set is a just an attribute table with some additional metadata. The metadata associated with a parameter set is the same as a script. Parameter sets can be accessed by specifying one or more of the given database parameters:

- Category – the system that 'owns' the parameter set. The convention is to use the system tag (e.g. atst.tcs, atst.ocs, atst.ics.vbi, etc.) as the category.

- Name – the identifying name for the parameter set within the category. In the case of parameter sets use by instruments, this is typically the observation mode (e.g. observe, gain, setup, etc.).

- ID – a distinguishing tag for different parameter sets within a given category and name. For example, the parameter sets within the atst.ics.vbi category and named observe for the Observe operation mode may have ids such as generic, tableA, tableB, etc.)

- Version – unique key for any parameter set. Regardless of category, name, and id, all parameter sets are automatically tagged with a unique version number. The version number is updated whenever a parameter set is inserted or updated in the parameter set database.

For a complete list of script storage/accessor methods see the source or api documents for atst.cs.services.ParamaSetDB (Java) or atst::cs::services::ParamSetDB (C++).

11 COMPONENTS

The **Component** is the foundation for all applications in ATST. Most ATST applications extend the **Controller**, a subclass of **Component** that adds *configuration-management* features. Components are managed by **Containers**, which are responsible for managing the *lifecycle* characteristics of components. Consequently, there are no **main** functions for components – components do not exist as standalone entities. Containers also provide components with access to the services and tools described in earlier sections.

11.1 COMPONENT LIFECYCLES AND FUNCTIONALITY

The ATST *Container/Component Model* (CCM) distinguishes between the *lifecycle* of a component and the *functionality* provided by a component. The lifecycle of components is consistent across ATST. It is this consistency that allows components to be managed by ATST containers. Containers can manipulate the lifecycle characteristics of any component, without regard to the functional behavior. The functionality of a component, on the other hand, is unique to that component and implements the needs required by ATST of that component. Of course, some components share many common characteristics. ATST software developers are generally expected to use derivatives of components (i.e., controller).

This split of lifecycle and functional characteristics has three key advantages:

- The lifecycle characteristics can be implemented once, by CSF,
- The necessary overlap between lifecycle behavior and functional behavior occurs in well-defined places, and
- Component developers can concentrate exclusively on implementing the functional behavior.

11.2 COMPONENT LIFECYCLE

The section on containers outlines the steps performed by a component's container as part of the lifecycle management of the component. This section covers these steps from the component developer's perspective and identifies places where functional behavior can be injected into the lifecycle operations. Additional lifecycle characteristics of components are also covered here.

11.2.1 Creation

A container creates a component by invoking that component's **default** constructor. At this time none of the CSF services are available to the component – no log service, property service, etc. For this reason, developers should avoid putting any functionality into the default constructor. In fact, the *ideal* constructor is *empty*! A component that has been created but not initialized is said to be *loaded*.

11.2.2 Initialization

The next step is *initialization*. Initialization is the process of preparing a component for operation, but stops short of starting that operation. Typical steps taken during initialization include:

- *Metadata* about the component's attributes is loaded, and
- Any special memory needs (fixed buffers, memory maps, etc.) are satisfied.

The first of these is performed automatically by CSF (though it may be deferred through lazy evaluation, that deferral is transparent to the operation of the component). The latter is an action based upon the functional behavior required of the component and so must be done by the component developer.

Under no circumstances should a component move any physical mechanisms during initialization. The component is not yet running and is not available for access by other components at this time.

Before initializing the component, the container creates a toolbox for the component, populates it with service helper tools, and attaches the toolbox to the container. The container also binds a *name* to the component at this time, but does **not** register this name with the ATST connection service (yet). The container then calls the component's **init()** method. Any component initialization that isn't performed by the framework should be added to the **doInit()** method by the component developer. The CSF services are available for use by component developers to assist in this part of initialization.

11.2.3 Startup

Once the component has been initialized, the container (and hence the component) waits for a directive to start the component operating. When this directive is received, the container invokes the component's **startup()** method. Upon completion of this method, the component is assumed to be *running* and ready to accept *functional directives*. The container then announces the availability of the component by registering the component's name with the ATST connection service and informing the Container Manager.

It should be pointed out that "ready to accept functional directives" does **not** mean ready to accept *any* functional directive. The behavior of a component that has completed startup successfully is defined by the functional definition of that component.

11.2.4 Operation

Very little lifecycle activity takes place during component operation (i.e., while it is *running*). There are two essential tasks that are performed:

- Monitoring component health and
- Controlling logging (*especially* debug).

While a component is running, the ATST Health Service publishes a *heartbeat* event at a regular interval. This heartbeat includes the current **health** of the component. The component's container monitors the heartbeat and reports any irregularities to the Container Manager. From a component developer's view, all that is needed is to keep the component health information up-to-date. (It should be noted, however, that a heartbeat in a *multithreaded* process cannot be treated as an accurate report on *all* threads. It may well be the case that the *only* functioning thread is the one reporting the heartbeat. For this reason, the heartbeat is perhaps best viewed as a monitor of the network connection and the host hardware, and not seen as a monitor of the software itself.)

Logging control – enabling/disabling categories and changing the debug levels of categories (see the section on the Log Service) – is another lifecycle management activity performed during component operation. This is handled entirely by the Log Service and needs no component developer action. However, a component developer can test whether or not a category is enabled and check the current debug level. Decisions in the developer's code may then be based on the results of those actions.

11.2.5 Shutdown

Near the end of its lifetime a component may be *shut down* by its container by calling the **shutdown()** method. This makes the component unavailable for functional access. That is, the shut down process is the inverse of the startup step. The component developer is responsible for safely undoing all actions they introduced during start up (the common services, in conjunction with the base **Component** class undo their own actions). If a container is directed to restart a shut down component, the startup actions are repeated.

Besides restarting, the only other operation available on a shut down component is the *un-initialization* of the component.

11.2.6 Un-initialization

When a Component has been shut down, it is back in the *initialized* lifecycle stage. The component can then be moved back to the *loaded* stage by calling the **uninit()** method. This operation is the inverse of initialization. Typically, the next stage is the *removal* of the component from the container. It is also possible to reinitialize the component.

11.2.7 Removal

Components that are *transient* may be removed once they have been shut down. The **remove()** method is called to allow the release of any resources acquired by the Component. The **remove()** is thus the inverse of the loading a component. Removing a component removes it from the container and from the ATST Connection Service. No action can be taken with that component from then on – although a new component may be created in its place.

11.3 FUNCTIONAL ARCHITECTURE

While a component is operating, there are only two basic functional operations that are available. (Subclasses, of course, define more operations). Another application connected to a component may:

- Request the values of attributes and
- Set the values of attributes.

Both operations are subject to the standard ATST access policy. Controller components extend this functionality by adding support for managing configurations. See the section on controllers for details. Custom components extend this functionality in other ways on a case-by-case basis.

Component developers must also provide support for determining the *health* of the component's behavior by implementing code used by the Health Service when checking system health. This support is described in the language-specific sections below.

11.4 SIMULATED COMPONENTS

ATST Components may be *simulations*. A simulated components is not permitted to submit configurations to controllers and has all outgoing events tagged as coming from a simulated component. A single instance of a component is not permitted to switch between simulated and non-simulated. Switching is performed by unloading one instance from its Container and loading another.

Once a component has been marked as a simulated component, all outgoing events contain an attributed named __.**simulated** with the value **true**. The Common Services Framework adds this attribute automatically to the event. Similarly, CSF enforces the prohibition on having simulated components submit configurations to controllers.

11.5 JAVA-BASED COMPONENTS

The base Component class **atst.cs.ccm.component.Component** is an abstract class that must be sub-classed. This base class has very little support for the ATST functional architecture. Component developers must add functionality to a subclass.

There are *hooks* in the base class for adding functional behavior during the deployment steps described above. Such behavior is added by overriding one or more of the following methods:

```
// functionality needed during Component initialization
void doInit();

// functionality needed during Component startup
void doStartup();

// functionality needed during Component shutdown
void doShutdown();

// functionality needed during Component un-initialization
void doUninit();

// functionality needed during Component removal
void doRemove();
```

In **doInit** and **doStartup** the arguments passed in via **args** are determined by the role the Component plays in ATST. Often, **args** is **null** in both instances.

Developers may throw an **atst.cs.ccm.component.LifeCycleChangeException** from any of the above methods. This exception is caught and the exception message and stack trace are logged. In the cases of **doInit** and **doStartup**, throwing the exception also aborts the lifecycle change. It does not abort the change when thrown from **doShutdown**, **doUninit**, or **doRemove**.

The two functional operations are implemented as:

```
// request values of Attributes from Component
IAttributeTable get(IAttributeTable params);

// set values of Attributes in Component
void set(IAttributeTable params);
```

The first of these uses the attribute names in **params** to identify attributes within the component that are included in the return value. Attributes with unknown names are removed from the table. Attributes with null or empty string values returned from the get are left unmodified. In other words, if the Property or Cache value for an attribute in the table is null or empty string, then that attribute will not be modified; the value previously in the table, before the get(…), will remain. The second sets the component's attributes to the respective values. Attributes are simply passed to the **doSet** method described below. Functionality, including the actual setting of values, must be added by overriding the **doSet** method.

Subclasses of component should **not** override the above **get** and **set** operations. Instead, subclasses should override the respective methods:

```
void doGet(IAttributeTable params);
void doSet(IAttributeTable params);
```

Some component subclasses may want to do addition processing when the log debug level changes. Two convenience methods may be overridden to provide for additional functionality at that point:

```
// defaults to an empty call
void doSetDebugLevel(int level);

// defaults to an empty call
void doSetDebugLevel(String category, int level);
```

11.5.1 Simulated Java Components

A Java-based component may be marked as simulated. This enables the enforcement of the functional restrictions imposed on simulated Components.

```
// mark this Components as simulated
void markAsSimulated();
```

This method should be called as early as possible. If possible it should be embedded in the component's *default constructor*, as in:

```
public MySimComponent() {
    markAsSimulated();
}
```

or in the unnamed initializer:

```
{
   markAsSimulated();
}
```

All simulated Components must be marked in this manner.

11.5.2 Java Example Component

See Controller Example (section 12.7.2) for an example controller. By changing the super class from **atst.cs.controller.Controller** to **atst.cs.ccm.component.Component**, and removing the methods related to actions, you have a fully functional component.

11.6 C++-BASED COMPONENTS

The base component class **atst::cs::ccm::component::Component** is an abstract class that must be sub-classed. This base class has very little support for the ATST functional architecture. Component developers must add functionality to a subclass.

There are *hooks* in the base class for adding functional behavior during the deployment steps described above. Such behavior is added by overriding one or more of the following methods:

```
// Functionality needed during Component initialization
void doInit();

// Functionality needed during Component startup
void doStartup();

// Functionality needed during Component shutdown
void doShutdown();
```

In **doInit** and **doStartup** the arguments passed in via **args** are determined by the role the component plays in ATST. Often, **args** is zero in both instances.

Additional functionality may be added by overriding the following methods:

```
// functionality for Component uninitialization
void doUninit();

//functionality for Component removal
void doRemove();
```

The two functional operations are implemented as:

```
// Request values of Attributes from Component
pIAttributeTable get(pIAttributeTable params);

// Set values of Attributes in Component
void set(pIAttributeTable params);
```

The first of these uses the attribute names in **params** to identify attributes within the component that are included in the return value. Attributes with unknown names are ignored. The second sets the component's attributes to the respective values. Attributes are simply passed to the **doSet** method described below. Functionality, including the actual setting of values, must be added by overriding the **doSet** method.

Subclasses of component should not override the above **get** and **set** methods. Instead, subclasses should override the respective methods:

```
void doGet(pIAttributeTable params);

void doSet(pIAttributeTable params);
```

More details can be found in the sections on predefined subclasses of **atst::cs::ccm::component::Component** such as **Controller**.

11.6.1 Simulated C++ Components

A C++-based component may be marked as simulated. This enables the enforcement of the functional restrictions imposed on simulated Components.

```
// mark this Components as simulated
void markAsSimulated();
```

This method should be called as early as possible. If possible it should be embedded in the component's *default constructor*, as in:

```
public MySimComponent() {
    markAsSimulated();
}
```

All simulated Components must be marked in this manner.

11.6.2 C++ Example Component

See Controller Example (section12.9.2) for an example controller. By changing the super class from **atst::cs::controller::Controller** to **atst::cs::ccm::component::Component**, and removing the methods related to actions, you have a fully functional component.

12 CONTROLLERS

A **Controller** is a subclass of a component, used to manipulate configurations. The Controller class is only one possible way of manipulating components. However, a controller is ideally suited for many situations, especially those that need to handle multiple, simultaneous configurations.

A component does nothing with configurations; it simply manages its own lifecycle and accepts low-level **get** and **set** operations on attribute tables. Since a configuration is more than a grouping of attribute-value pairs, there needs to be a class that controls configuration lifecycle issues. Hence, the **Controller** class exists. Some useful subclasses of **Controller** for control include the **Base Controller**, **Management Controller** and various subclasses of the **Hardware Controller**.

Controllers are part of the *application framework* layer of the ATST Common Services Framework.

12.1 CONTROLLER FEATURES

12.1.1 Command-Action-Response

The controller implements a command-action-response model. In this model, commands are submitted to the controller where they are either accepted or rejected based upon the validity of the argument and the state of the controller. If a command is accepted by the controller it causes an independent action to begin. A response to the command is returned immediately. The action begins matching the current configuration to the new demand configuration. When the configurations match (i.e., the controller has performed the input operations) the action signals the successful end of the action. If the configurations cannot be matched (whether by hardware failure, external cancel command, timeout, or some other fault) the action signals the unsuccessful end of the action.

The important features of the command/action/response model are:

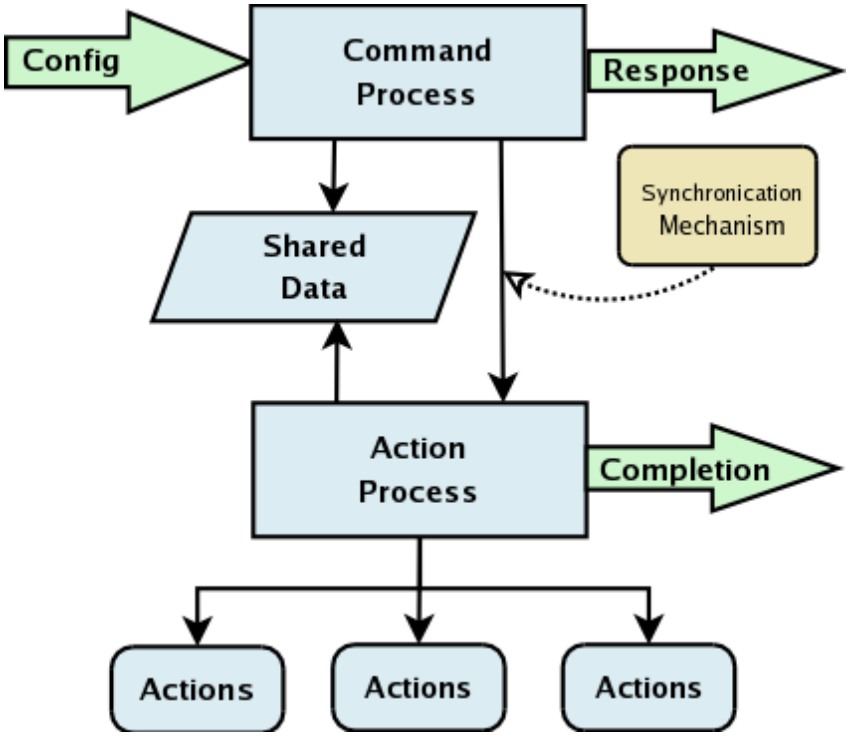
- Commands are never blocked. As soon as one command is submitted, another one can be issued. The behavior of the controller when two or more configurations are submitted can be configured on a per-controller basis.

- The actions are performed using one or more separate threads. They can be tuned for priority, number of simultaneous actions, critical resources, or any other parameters.

- Action completions produce events that tell the state of the current configuration. Actions push the lifecycle of the configuration through to completion.

- Responses may be monitored by any other Components/Controllers.

ATST controllers use threads to implement both command and action processing.



12.1.2 Command Thread

The command thread receives configurations from the external interface and performs several basic sanity tests on them. In addition to checking for properly formed configurations, the command thread calls the Property Service to test individual attribute's ranges and types. The command thread also calls a **doSubmit** method provided by a subclass to test for other conditions that might preclude executing the configuration.

Once these tests are performed the command thread queues the configuration and signals the action manager. At this point a response is returned to the external source of the command. The response is an integer representing the result of the submission. Its value will be one of:

- **OK (0)** – The configuration has been accepted for action. This is the *only* code that will result in the configuration being scheduled for action.
- **BUSY (-1)** – The configuration has been rejected because the controller cannot perform any additional simultaneous actions and cannot queue the submitted configuration.
- **BAD_PARAM (-2)** – The configuration has an invalid parameter value.
- **MISSING _PARAM (-3)** – The configuration has an invalid parameter value.
- **INCONSISTANT_PARAM (-4)** – The configuration has an invalid parameter value.
- **EXCEPTION (-5)** – There is a runtime error in the submit code for the target controller.
- **NOT_RUNNING (-6)** – The target controller is not at its *running* stage.

- **DUPLICATE (-7)** – The configuration ID matches one already being acted on.
- **NO_CONFIG (-8)** – There was no configuration submitted.
- **SIMULATED (-9)** – The request came from a component running in *simulation* mode.
- **REJECTED (-10)** – The configuration failed the "can run" check.
- **ERROR (-11)** – The action had an unexpected error that caused the action to terminate.
- **ABORTED (-12)** – The action was externally aborted and failed to complete.
- **CANCELED (-13)** – The action was externally cancelled and failed to complete.
- **INTERLOCKED (-14)** – The action was stopped because of an interlock.

The command thread also handles the **cancel** command. The configuration identified by the argument to **cancel** is either queued or active. If the former, it is removed from the queue and an **abort** event is sent for it. It is then destroyed. If the configuration is currently active, the action thread it is running under is issued an **abort** signal, whereupon it propagates the abort command to any subsystems involved in the processing, then aborts and destroys the configuration. The **doCancel** command is available for subclasses to implement their own behavior when the **cancel** command is received.

The command thread also handles the **abort** command. The configuration identified by the argument to **abort** is either queued or active. If the former, it is removed from the queue and an **abort** event is sent for it. It is then destroyed. If the configuration is currently active, the action thread it is running under is issued an ~~**cancel**~~**abort** signal, whereupon it propagates the ~~cancel~~ abort ~~command~~ to any subsystems involved ~~in the processing~~, then aborts and destroys the configuration. The **doAbort** command is available for subclasses to implement their own behavior when the **abort** command is received. The cancel command is handled in the same manner. CSF does not prevent an action from receiving an abort signal after it ias received a cancel signal (or vice-versa).

Finally, the command thread also handles the commands for **pause** and **resume**. The **pause** command either keeps the configuration from leaving the queue or forces an action thread to pause the configuration and all of the associated subsystems. The **resume** command causes either the queue or action thread to restart executing the configuration. Pausing an active configuration is the responsibility of subclasses which may implement the **doPause** and **doResume** commands to do so.

12.1.3 Action Manager

The action manager is a thread responsible for managing the execution of the incoming configurations. The actual implementation of the action manager is beyond the scope of this document, but it is useful to understand its basic operation. Configurations are queue in priority order based upon their *start time*. If a configuration requires immediate execution, the action manager finds an available action thread and assigns the execution details to that thread. If a configuration has a **csf:startTime** attribute the action manager delays execution until the requested time. A configuration may also contain a **csf:startDeadline** and or a **csf:timeout** attribute. Both attributes provide a time by the action must be completed. If the action has not started by the deadline, it is aborted. The **csf:timeout** attribute is the amount of time to allow the action to run before aborting it because it is assumed to have failed. If a **csf:startTime** attribute is present, the completion deadline imposed by the timeout is taken relative to the start time. If there is no **csf:startTime** attribute, the completion deadline is taken relative to the time that the action was scheduled

(shortly after the action has been submitted). The **csf:timeout** attribute may have a default value defined as a property of the controller. A timeout of 0 indicates no timeout.

The action manager may be configured on a per-controller basis for its behavior. Controllers that protect critical resources may have only one action thread. Queued configurations are held or aborted, depending upon the value of the **csf:fullThreadAction** attribute. Other controllers may have a large pool of action threads.

When an action thread completes it signals the action manager with a done or aborted signal using an *action callback*. The action manager then regenerates that signal as an event and deletes the configuration. The action thread is returned to the available thread pool.

Commands to **cancel**, **abort**, **pause**, or **resume** the execution of a configuration are passed through the action manager. If necessary, the action manager signals the appropriate action thread to perform the requested command.

12.1.4 Action Threads

Each controller has a pool of available action threads, the number may vary for each controller. Once taken from the pool by the action manager, an action thread is assigned a configuration. The heart of the action thread is the **action** command provided by a subclass of the **Controller** class. All action threads run the same **doAction** command, so it must be thread-safe. This command does not need to know about the lifecycle of the action thread, however, nor its interactions with the action manager. It only needs to perform work upon the input configuration. Upon completion of an action, the **doAction** command returns an **ActionResponse** object indicating whether or not the action was successfully or unsuccessfully completed. In the case of successful completion (the functional code was successful, or the action was canceled), done is returned. In the case of unsuccessful completion (the functional code had problem or, the action was aborted) aborted is returned. Aborting and canceling actions should be handled differently by the functional code which is why the former is considered a failure and the later is considered a success. In the case of an action receiving an abort command that action should terminate *as soon as is safe*. This means that a motor might stop in the middle of nowhere or a camera might stop mid frame. In the case of an action receiving a cancel command that action should terminate *at its earliest possible convenience.* This means that a motor may continue to its next encoder position, or a camera may finish a frame or group of frames.

12.1.5 Action Callbacks

When a controller submits a configuration to another controller (for example, when the TCS submits a configuration to the MCS), a *callback* is attached to the submission to provide for action response synchronization. The callback provides two basic commands: **done** and **abort** for use by the action callback as signals. Developers may add functionality to these commands by implementing the **doDone** and **doAbort** commands. A third command **report** (and the associated **doReport**) may be used to issue *progress reports* during the processing of an action.

12.1.6 Simulated Controllers

Controllers, just like components, may be simulations. A simulated controller is not permitted to submit configurations to *any* other controllers and has all outgoing events tagged as coming from a simulated controller. A single instance of a controller is not permitted to switch between simulated and non-simulated. Switching is performed by unloading one instance from its container and loading another.

Once a controller has been marked as a simulated controller, all outgoing events contain an attributed named **simulated** with value **true**. This attribute is added automatically to the event by CSF. Similarly, CSF enforces the prohibition on having simulated controllers submit configurations to other controllers. For details on marking a controller as simulated see the sections on marking java and c++ components as simulated.

12.2 CONTROL OF CONFIGURATION LIFECYCLE

A configuration's lifecycle is well-defined in its definition; it is initialized, is running, and is completed. Each of these stages is entered through some type of external transition. The type of transition determines the event posted about the configuration to any interested parties (such as the OCS). Configuration lifecycle transitions within a controller are handled entirely by the base controller class. However, the base controller implementation only posts configuration lifecycle transition events on **done** or **aborted** transitions unless the **tcsf:traceconfigs** property/attribute has been set to *true*.

12.2.1 Scheduled

When a configuration arrives in the controller though the **submit** interface it is in the initialized state. The controller generates the "scheduled" event through the component interface to signal that it has accepted a configuration and is scheduling it for execution. The implementation details of the scheduler are not important for this discussion; it is enough to say that the scheduled phase of the configuration lifecycle may be arbitrarily short or long.

12.2.2 Running

Once the configuration is assigned to an action process things begin to happen. First, the state of the configuration goes to "running". Next, the action process begins to match the current and demand configurations. During this time other events (like position) are generated.

12.2.3 Completed

A configuration that is no longer being matched by the controller is "completed". An event is generated indicating whether the matching was successful ("done") or unsuccessful ("aborted") and the configuration is removed from the action process and destroyed.

A configuration may be completed successfully for a number of reasons:

- The desired state was successfully met.
- An external **cancel** command was issued for the configuration ID matching that of the configuration and the controller was able to gracefully end execution.

A configuration may not be successfully completed for a number of reasons:

- It was rejected prior to scheduling.
- It could not be scheduled.
- An external **abort** command was issued for the configuration ID matching that of the configuration.

- The configuration's **timeout** value was exceeded.
- The controller's action process determined the configuration could not be met.
- The command was stopped because of an interlock being raised
- The interlock status was high when the command was removed from the schedule and attempted to begin executing.

A component owned by the controller and involved in matching the configuration aborted its own configuration.

12.3 ACTION PROCESS

The above describes all of intracaties of an action, but in most cases it is not that complicated. The following describes the steps that a basic configuration takes in the action process. When a configuration is submitted to a Controller there are some parts of the process that are done automatically by CSF some parts done automatically by the csf and the controller and other parts that can or should be overridden for developers to add their own functionality. The following attempts to clearly define an action from start to finish.

Submit Thread: After a controller's submit method is called there are a few things that happen in the submit thread:

1. The Controller checks to make sure that it is running and rejects the configuration if it is not.
2. The Controller checks to see if it is already executing an action based on the config id of the submitted configuration and rejects the new configuration (as a duplicate) if it is.
3. The Controller checks each attribute passed in to make sure that it is properly typed, and in range. If *any* attribute is improperly typed, or out of range the configuration is rejected (as bad param).
4. The Controller calls its own `doSubmit()` to do any functional validation of the configuration. *This method is intended to be overridden by developers.* If `doSubmit()` returns an error, the configuration is rejected. By default the `doSubmit()` returns OK.
5. The Controller places the configuration on the Action Manager's queue.
6. The Action Manger checks to see if it is supposed to schedule configurations OR if there are free threads to carry out the new configuration. If not, the configuration is rejected (as busy).
7. The Action Manager calls the Controller's `getNewAction()` method and places the action on to the queue. *This method may be overridden by developers. A custom action can be useful as a pass box to allow information to be passed between the various methods carrying out the rest of the action.*
8. The Action Manager notifies the scheduler thread that something has been inserted into the queue.

Scheduler Thread: After the action has been placed on the queue the scheduler thread is responsible for assigning the action to a thread, and beginning its execution.

1. The scheduler attempts to acquire a free thread, and a configuration whose start time has passed. As soon as both are encountered, the scheduler calls the Controller's `canRun()` method.

2. The Controller's `canRun()` method checks to see if the controller is currently interlocked, and if so rejects the configuration.

3. The Controller calls its own `doCanRun()` to perform any functional checks on the configurations ability to run. *This method is intended to be overridden by developers. They should check that given the current state of the controller, the configuration can be matched.* If `doCanRun()` returns an error, the configuration is rejected. By default `doCanRun()` returns OK.

4. The scheduler calls the controller's `doSerialAction()` method. *This is intended to be overridden by developers. Because it is run by the scheduler thread it allows multi-threaded controllers to carry out portions of their actions in a serial manner. It is often not needed.* If `doSerialAction()` does not return OK, the Action fails with an Error. By default `doSerialAction()` is a no-op that returns OK.

5. The scheduler assigns the action to a thread and starts the thread.

Action Thread: After the scheduler has started the action thread, all remaining work is done on that thread. This is where any long-running parts of the action should take place.

1. The Action Thread calls the controller's `doAction()` method. *This method MUST be overridden by developers. This is where most of the work should get done.* If `doAction()` does not return OK, the Action fails with an Error. Be default `doAction()` is a not op that returns an **Error**.

12.4 INTERFACE

The **Controller** class has a public and protected interface. The public interface has an associated interface definition and communications implementation. The protected interface allows up-calls from the action task or subclasses.

12.4.1 Public Interface

In addition to the methods defined in the component public interface, the public interface for a controller adds:

- **submit** — Schedule a configuration to be executed. Returns **OK** (`0`) if the configuration can be scheduled and a non-zero flag otherwise (see below for a list of the known error flags).

- **cancel** — cleanly stop the execution of a scheduled configuration.

- **abort** — immediately stop the execution of a scheduled configuration

- **pause** — pause the execution of a scheduled configuration.

- **resume** — resume the execution of a paused configuration.

12.4.2 Protected Interface

The public interface methods are predefined by the base controller class and cannot be overridden. However, each makes calls to some protected methods that can be overridden:

- **doSubmit** — Subclass checks during a **submit** command. Returns **OK** (0) only if the configuration action can be *scheduled*. Only non-transient factors should be considered in this check. For example the status of an interlock is a transient factor because even if it is high now it might be low when the action attempts to execute.

- **doCanRun** – Subclass checks to see if the can be executed right now. Transient factors may be considered in this check. It is immediately before this method is called that the technical architecture looks at the interlock status in its determination of whether or not that action can start.

- **doAction** — Subclass performs the action indicated by the configuration. Returns an **ActionResponse** when the configuration action was successfully completed.

- **doCancel** — Subclass checks during a **cancel** command. Returns **true** if the configuration action can be cancelled.

- **doAbort** — Subclass checks during an **abort** command. Returns **true** if the configuration action can be aborted.

- **doPause** — Subclass checks during a **pause** command. Returns **true** if the configuration action can be paused.

- **doResume** — Subclass checks during a **resume** command. Returns **true** if the configuration action can be resumed.

More details about the above methods can be found in the language-specific sections that follow.

12.4.3 Attributes

Attributes of a controller can be read and modified through the *get* and *set* commands in the Component interface.

- **csf:threadModel** — whether the pool of action threads is **fixed** or **growable**.

- **csf:numThreads** — the current number of action threads.

- **csf:maxThreads** — the upper bound on the number of action threads, when **growable**.

- **csf:fullThreadAction** — queue or reject configurations when there are no available threads.

- **csf:activeThreads** — the number of active action threads.

- **csf:schedList** — a list of configuration IDs in the schedule.

- **csf:timeout** — the default action timeout if not given in the configuration.

- **csf:traceConfigs** — determines whether or not configuration lifecycle events are posted on non-terminal transitions (**DONE** or **ABORTED** transitions always have status events posted).

- **csf:minActionDelay** — the minimum allowed delay (in milliseconds) when pausing an action.

- **csf:scheduleCheckRate** — the delays between checks of the Action queue for runnable actions.

There are attributes in a configuration that the technical architecture uses to run the configuration.

- **csf:timeout** — the time limit (in *milliseconds*) to be imposed on the action to match the configuration.

- **csf:startTime** — the earliest time at which the action should begin **`(yyyy/mm/dd:hh:mm:ss.s tz or yyyy/DDD:hh:mm:ss.s tz,`** where **`DDD`** is the Julian day). This is the string form of an **AtstDate**.

- **csf:startDeadline** — the time the action should be started by (**`yyyy/mm/dd:hh:mm:ss.s tz or yyyy/DDD:hh:mm:ss.s tz`**, where **`DDD`** is the Julian day). This is the string form of an **AtstDate**.

Here **csf:timeout** is the time limit imposed on the action *from the moment it actually starts execution*. It does not include time that the action is spent queued waiting to start. If omitted, the default timeout defined by the specific controller is used. A value of `0` implies no limit. Conversely, **csf:startDeadline** is the point at which queued action is not longer to be considered available for execution. If omitted, then the action will remain queued until executed or forcibly removed from the queue.

12.4.4 Events

A controller generates the following events. All generated event names are prefixed with the name of the specific controller instance (see example above). The following events are used:

- **configstate** – the status of the action on a configuration by this controller. The event includes the *configuration ID*, possibly a reason for the reported status (in the case of an *aborted* status), and the new *state* of the configuration, one of:

    - **SCHEDULED** – the action has been scheduled
    - **RUNNING** – the action is actively running
    - **PAUSED** – the action has been paused
    - **ABORTED** – the action has completed unsuccessfully
    - **DONE** – the action has completed successfully

In practice, only the **ABORTED** and **DONE** states are normally reported. Note however, that controller developers do not generate these events; they are generated automatically.

12.5 ACTION CALLBACK INTERFACE

The action callback also has public and protected interfaces. The implementation of the public interface methods is complete and provided as part of the Common Services Framework. Controller developers can add additional functionality by implementing the methods given in the protected interface.

12.5.1 Public Interface

- **done** — reports successful completion of a submitted configuration's match
- **abort** — reports unsuccessful completion of a submitted configuration's match
- **report** — reports on the status of the action on a submitted configuration

12.5.2 Protected Interface

- **doDone** — action response to successful submitted action completion.
- **doAbort** — action response to unsuccessful submitted action completion.
- **doReport** — action status report

12.5.3 Controller Properties

Properties should exist for all of the properties mentioned in section 12.4.3. If a controller is capable of carrying out multiple simultaneous actions, the thread model should be set to **growable**. In addition to defining the acceptable and default value of each of those properties, controller specific attributes also need associated metadata. For example a controller that takes a 'mode' attribute should define the acceptable values of that attribute. See TN-0120 for information about tools used to modify the persistent store.

Controller properties are organized by controller *name*, not by controller *class names*. This means that different controllers implemented using the same class, be it C++ or Java, may have different property metadata. It also means that these properties must be maintained in the Property Service persistent store by controller name.

12.6 JAVA-BASED CONTROLLERS

12.6.1 The Public Interface

Controllers all extend the base **atst.cs.controller.Controller** class and implement the **atst.cs.interfaces.IController** interface. The following constants methods are defined by that interface in addition to those methods inherited from the **atst.cs.interfaces.IComponent** interface:

```
// response codes for submit/doSubmit
int OK, BUSY, BAD_PARAM, MISSING_PARAM, INCONSISTENT_PARAM, EXCEPTION,
      NOT_RUNNING, DUPLICATE, NO_CONFIG, SIMULATED;

// match the supplied Configuration
int submit(IConfiguration config);

// cancel processing of the named Configuration
void cancel(String configID);

// abort processing of the named Configuration
void abort(String configID);

// pause processing of the named Configuration
void pause(String configID);
```

```
// resume processing of the named Configuration
void resume(String configID);
```

In addition, the following convenience methods are available as part of the public interface:

```
// match the supplied configuration, invoking the supplied
// callback in the caller on action completion.
int submit(IConfiguration config,
         IActionCallback callback);
```

The last method is the preferred interface methods as it automatically ensure that no race condition exists between the submission and the subscription of the callback. In addition, it ensures that the callback will only be invoked on events originating from the action that result from that specific submission.

All of these methods are fully implemented by the base Controller class supplied as part of the CSF. Controller developers may add functionality at these controller lifecycle stages by implementing the appropriate methods defined in the *protected interface* below.

12.6.2 The Protected Interface

The lifecycle protected methods **doInit**, **doStartup**, **doShutdown**, **doUninit**, and **doRemove** found in the Component class are also provided in the Controller class. These methods may also throw the **atst.cs.ccm.component.LifeCycleChangeException** described in the Component section.

Additional functionality can be added by controller developers by overriding the following Controller methods:

```
// validate the non-transient factors of the configuration
ActionResposne doSubmit(IConfiguration config);

// Submits a Configuration immediately after doStartup
IConfiguration doRunning();
// validate the transient factors of the configuraton
ActionResponse canRun(IConfiguration config);

// perform any serial parts of an action
ActionResponse doSerialAction(IConfiguration config);

// perform the action
ActionResponse doAction(IConfiguration config);

// cancel processing of the named Configuration
boolean doCancel(String configID);

// abort processing of the named Configuration
boolean doAbort(String configID);

// pause processing of the named Configuration
boolean doPause(String configID);

// resume processing of the named Configuration
boolean doResume(String configID);
```

The error code in **doSubmit**'s **ActionResponse** is returned by **submit**. The methods **doCancel**, **doAbort**, **doPause**, and **doResume** are only called if the action on the identified configuration is currently active. If the action is scheduled, but not yet running, it is handled internally. Each should return true if the action was successfully cancelled, aborted, paused, or resumed, respectively. Unless overridden, each of the above do nothing, so the default behavior is that executing actions may not be paused, resumed, aborted, or cancelled.

The result of evaluating **doAction** should be **Controller.ACTION_OK** only if the action is successfully completed. If the action was unsuccessful an error **ActionResponse** response should be returned. See the API for **atst.cs.controller.ActionResponse** for details

12.6.3 Action Callbacks

All action callbacks extend **atst.cs.controller.ActionCallback** and implement the **atst.cs.interfaces.IActionCallback** interface. The following methods are defined by this interface:

```
// report successful matching of the submitted Configuration
void done(IAttributeTable data);

// report unsuccessful matching of the submitted Configuration
void abort(IAttributeTable data);

// report on the current status of the submitted Configuration
// (does not terminate the action).
void report(IAttributeTable data);
```

All of these methods are implemented by the **ActionCallback** base class and cannot be overridden. Controller subclasses can add additional operations to be performed by on successful completion of a submitted configuration by overriding the following methods from the **ActionCallbackAdapter** class:

```
// handle successful matching of the Configuration
void doDone(IAttributeTable data);

// handle unsuccessful matching of the Configuration
void doAbort(ActionResponse response, IAttributeTable data);

// handle a status report on the Configuration action
void doReport(String reason, atst.cs.interfaces.IAttributeTable data);
```

12.7 WRITING JAVA-BASED CONTROLLERS

This section describes the basic steps involved in writing a custom, Java-based controller. It introduces some of the support that is available to developers when they subclass **atst.cs.controller.Controller**, and offers some suggestions on how to handle common situations.

All ATST controllers extend the **atst.cs.controller.Controller** class. Controllers that want to submit configurations to *other* controllers should also construct callbacks that extend the class **atst.cs.controller.ActionCallback**. The next few sections describe the key methods that should be overridden by subclasses of **Controller** and **ActionCallback**. Naturally, additional support methods may also be added as needed.

12.7.1 The ControllerAdapter Class and Source File

The class **atst.cs.controller.ControllerAdapter** subclasses **atst.cs.controller.Controller** and includes rudimentary implementations of the methods in the protected interface where you can attach application specific functionality to a controller subclass. While **ControllerAdapter** itself may be subclassed, a more practical use is to use its *source code* as a template for constructing your own controller subclass:

1. Copy **`$ATST/src/java/atst/cs/controller/ControllerAdapter.java`** into the source directory for your new controller subclass, renaming it to your new controller name.

2. In that new source file, change:

   - the package name to your package,

   - the class name to your controller subclass name.

3. Now edit that source file to introduction the functionality required for your application.

12.7.2 Controller Example

The following is an example of a very simple controller. In short, the controller implements a pseudo-state machine. The controller's health is related to the current state of the machine and only certain state changes are permitted.

```
package atst.cs.controller;

import java.util.Random;

import atst.cs.ccm.component.LifeCycleChangeException;
import atst.cs.data.Attribute;
import atst.cs.data.AttributeTable;
import atst.cs.interfaces.IAttributeTable;
import atst.cs.interfaces.IConfiguration;
import atst.cs.services.Event;
import atst.cs.services.Health;
import atst.cs.services.Log;
import atst.cs.util.Cache;
import atst.cs.util.Misc;

/**
 * The example controller is a simple state machine. That takes a random
 * amountof time to transition between states. The state is only allowed
 * to transition up. To transition down, the controller must be shutdown,
 * or an action must be aborted, or interlocked.
 *
 * @author John Hubbard (jhubbard@noao.edu)
 */
public class ExampleController
      extends Controller {

   private static final int      AT_REST_STATE     = 0;
   private static final int      UNKNOWN_STATE     = -1;
   private Random                rand              = null;
   private PostingThread         pt                = null;
   int                           state             = UNKNOWN_STATE;
```

```
/*
 * Controller support methods
 */

@Override
protected ActionResponse doSubmit(IConfiguration config) {
   // if we get here we know that the technical architecture has
   // checked and the attributes are properly typed and
   // within range we just need to make sure that there is a
   // new state attribute
   if (!config.contains("state"))
      return ActionResponse.missingParam("state");
   // and that min time is less than max time
   int min = config.contains("minSimTime") ? config
         .getInteger("minSimTime") : Cache.lookupWithDefault(
         new Attribute("minSimTime", 500)).getInteger();

   int max = config.contains("maxSimTime") ? config
         .getInteger("maxSimTime") : Cache.lookupWithDefault(
         new Attribute("maxSimTime", 5000)).getInteger();

   if (min <= max) return ACTION_OK;
   // the params aren't bad they are just inconsistent
   else return ActionResponse.inconsistent("min must be < max");
}

@Override
protected ActionResponse doCanRun(IConfiguration config) {
   // make sure that the new state is a valid transition from the
   // current state.
   int newState = config.getInteger("state");
   if (newState > state) return ACTION_OK;
   else return ActionResponse
         .reject("New state isn't greater than current state.");
}

@Override
protected ActionResponse doAction(IConfiguration config) {
   Action a = getAction(config.getId());
   int newState = config.getInteger("state");
   long delay = calcDelay(config);
   long timeTaken = 0;
   // wait for the action to complete
   while (timeTaken < delay) {
      Misc.pause(50);
      timeTaken += 50;
      if (a.paused()) a.pause(25);
      if (a.interrupted()) {
         if (a.wasAborted()) state = UNKNOWN_STATE;
         else if (a.wasCanceled()) state = AT_REST_STATE;
         else if (a.wasInterlocked()) state = UNKNOWN_STATE;
         else {
            state = UNKNOWN_STATE;
            Log.warn("Unrecognized interrupt reason!");
         }
         return a.interruptReason();
```

```
            }
        }
        state = newState;
        return ACTION_OK;
    }

    private int calcDelay(IConfiguration config) {
        int min = config.contains("minSimTime") ? config
                .getInteger("minSimTime") : Cache.lookupWithDefault(
                new Attribute("minSimTime", 500)).getInteger();

        int max = config.contains("maxSimTime") ? config
                .getInteger("maxSimTime") : Cache.lookupWithDefault(
                new Attribute("maxSimTime", 5000)).getInteger();

        int delay = min;
        delay += rand.nextInt(max - min);
        return delay;
    }

    @Override
    protected boolean doAbort(String configId) {
        // this controller can always abort
        // nothing to do here
        return true;
    }

    @Override
    protected boolean doCancel(String configId) {
        // this controller can always cancel
        // nothing to do here
        return true;
    }

    @Override
    protected boolean doPause(String configId) {
        // this controller can always pause
        // nothing to do here
        return true;
    }

    @Override
    protected boolean doResume(String configId) {
        // this controller can always resume
        // Nothing to do here.
        return true;
    }

    /*
     * Lifecycle support methods.
     */

    @Override
    protected void doInit() throws LifeCycleChangeException {
        rand = new Random();
        pt = new PostingThread();
    }
```

```
@Override
protected void doStartup() throws LifeCycleChangeException {
   Misc.startDaemon(pt);
   state = AT_REST_STATE;
}

@Override
protected void doShutdown() throws LifeCycleChangeException {
   pt.stop();
}

@Override
protected void doUninit() throws LifeCycleChangeException {
   pt = null;
   rand = null;
}

// protected void doRemove(){}
// omitted because nothing needs to be done on remove

@Override
protected void doGet(IAttributeTable table) {
   if (table.contains("state"))
      table.insert(new Attribute("state", state));
   // getting all other attributes is handled by the technical
   // architecture or the Cache.
}

@Override
protected void doSet(IAttributeTable table) {
// setting of all other is handled by the technical architecture
// or stored in the Cache until it is needed.
}

private class PostingThread
      implements Runnable {
   private boolean      stopFlag;

   public PostingThread() {
      stopFlag = false;
   }

   @Override
   public void run() {
      while (!stopFlag) {
         Misc.pause(500);// post at 2 hz
         IAttributeTable table = new AttributeTable();
         table.insert(new Attribute("state", state));
         Event.post(getName() + ".cstate", table);
      }
   }

   public void stop() {
      stopFlag = true;
   }
}
```

```
}
```

12.7.3 The ActionCallbackAdapter Class and Source File

The class **atst.cs.controller.ActionCallbackAdapter** subclasses **atst.cs.controller.ActionCallback** and includes rudimentary implementations of the **doDone**, **doAbort**, and **doReport** methods in the protected interface where you can attach application functionality. It serves the same roles to the **ActionCallback** class as **ControllerAdapter** serves to the Controller class.

12.7.4 Action Callback Adapter Methods

This section covers the roles of the protected methods that can be overridden in **ActionCallbackAdapter** subclasses to handle action responses that result from submitting configurations to other Controllers. Applications submit configurations to Controllers using one of the **submit** methods defined in the **IController** interface after connecting to the controller.

If the application *does not care* whether the resulting action completes successfully or not (e.g. the application is part of a processing pipeline), it may use the method **submit(IConfiguration config)**. Otherwise, one of the submit methods that allows attaching a callback should be used.

All action callbacks should subclass **atst.cs.controller.ActionCallback**, which provides the core behavior required of action callbacks by the technical architecture. A few methods are available for overriding to add functional behavior. The three key **ActionCallback** methods: **doDone**, **doAbort**, and **doReport** are called with the current configuration and the value of the event that led to the invocation of the callback (as an **AttributeTable**). In addition, **doAbort** also takes an **ActionResponse** object containing the reason for the aborting. The **doReport** method also takes a string containing the reason for the report.

***Method** doDone*

The **doDone** method is called internally by the **done** method when the target controller reports successful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding controller (or simple component, potentially!) in most cases some type of access to the controller will be needed (to set synchronization flags, for example). The methods **getController()** and **getComponent()**, described below, provide this access.

***Method** doAbort*

The **doAbort** method is called internally by the **abort** method when the target controller reports unsuccessful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding controller (or simple component, potentially!) in most cases some type of access to the controller will be needed (to set synchronization flags, for example). The method **getController()**, described below provides this access.

***Method** doReport*

The **doReport** method is called internally by the **report** method whenever a controller wants to issue a progress report on the processing of an action. Its use is not required.

***Method** getController*

The convenience method **atst.cs.controller.Controller getController()** provides access to the controller that issued the command for the action being reported by this **ActionCallback**. The result will likely need to be upcast to the appropriate controller subclass to call getters and setters to access information that

needs to be shared between that controller and this callback. For example, the **doDone** method could be written as:

```
protected void doDone(IConfiguration config) {
    ((MyController)getController().signalDone(config.getId());
}
```

The method signals successful completion of this action to the controller. Here, the **signalDone** method has been added as a public method to the controller subclass **MyController**.

***Method** getComponent*

The convenience method **atst.cs.component.Component getComponent()** provides access to the Component that issued the command for the action being reported by this **ActionCallback**. The result will likely need to be upcast to the appropriate Component subclass to call getters and setters to access information that needs to be shared between that component and this callback. For example, the **doDone** method could be written as:

```
protected void doDone(IConfiguration config) {
    ((MyComponent)getComponent().signalDone(config.getId());
}
```

The method signal successful completion of this action to the component. Here, the **signalDone** method has been added as a public method to the Controller subclass **MyComponent**.

This method ***must not be used*** if the submitter is a Controller subclass. Use **getController()** (above) instead.

***Method** getReason*

This convenience method provides access to the reason reported in an *abort* or *report* action response. A null is returned until an abort or report response has occurred.

***Method** getData*

This convenience method provides access to any application-specific data (if any) that has been associated with the latest action *report* response. It returns null until a report has occurred.

***Method** getConfig*

This convenience method provides access to the Configuration that was associated with the action.

***Method** isDone*

This convenience method returns true if the action callback has reported completion of the action (successful or unsuccessful).

***Method** isAborted*

This convenience method returns true if the action callback has reported failure of the action.

***Method** waitForDone(long delay)*

This convenience method blocks until **isDone()** returns true.

12.8 C++-BASED CONTROLLERS

12.8.1 The Public Interface

Controllers all extend the base **atst::cs::controller::Controller** class and implement the **atst::cs::interfaces::IController** interface. The following constants methods are defined by that interface in addition to those methods inherited from the **atst::cs::interfaces::IComponent** interface:

```
// response codes for submit/doSubmit
int OK, BUSY, BAD_PARAM, MISSING_PARAM, INCONSISTENT_PARAM, EXCEPTION,
      NOT_RUNNING, DUPLICATE, NO_CONFIG, SIMULATED;

// match the supplied Configuration
int submit(pIConfiguration config);

// cancel processing of the named Configuration
void cancel(const std::string& configID);

// abort processing of the named Configuration
void abort(const std::string& configID);

// pause processing of the named Configuration
void pause(const std::string& configID);

// resume processing of the named Configuration
void resume(const std::string& configID);
```

In addition, the following convenience methods are available as part of the public interface:

```
// match the supplied configuration, invoking the supplied
// callback in the caller on action completion.
int submit(pIConfiguration config,
         std::tr1::shared_ptr<IActionCallback> callback);
```

The last method is the preferred interface methods as it automatically ensure that no race condition exists between the submission and the subscription of the callback. In addition, it ensures that the callback will only be invoked on events originating from the action that result from that specific submission.

All of these methods are fully implemented by the base Controller class supplied as part of the CSF. Controller developers may add functionality at these controller lifecycle stages by implementing the appropriate methods defined in the *protected interface* below.

12.8.2 The Protected Interface

The lifecycle protected methods **doInit**, **doStartup**, **doShutdown**, **doUninit**, and **doRemove** found in the Component class are also provided in the controller class. These methods may also throw the **atst::cs::ccm::component::LifeCycleChangeException** described in the Component section.

Additional functionality can be added by controller developers by overriding the following Controller methods:

```
// validate the non-transient factors of the configuration
pActionResponse doSubmit(pIConfiguration config);
```

```
// Submits a Configuration immediately after doStartup
pIConfiguration doRunning();

// validate the transient factors of the configuraton
pActionResponse canRun(pIConfiguration config);

// perform the serial portion of the action
pActionResponse doSerialAction(pIConfiguration config);

// perform the action
pActionResponse doAction(pIConfiguration config);

// cancel processing of the named Configuration
bool doCancel(std::string& configID);

// abort processing of the named Configuration
bool doAbort(std::string& configID);

// pause processing of the named Configuration
bool doPause(std::string& configID);

// resume processing of the named Configuration
bool doResume(std::string& configID);
```

The **doSubmit** method returns the same values as **submit**, shown above. The methods **doCancel**, **doAbort**, **doPause**, and **doResume** are only called if the action on the identified configuration is currently active. If the action is scheduled, but not yet running, it is handled internally. Each should return true if the action was successfully cancelled, aborted, paused, or resumed, respectively. Unless overridden, each of the above do nothing, so the default behavior is that executing actions may not be paused, resumed, aborted, or cancelled.

The result of evaluating **doAction** should be **Controller::ACTION_OK** only if the action is successfully completed. If the action was unsuccessful an error **ActionResponse** response should be returned. See the API for **atst::cs::controller::ActionResponse** for details

12.8.3 Action Callbacks

All action callbacks extend **atst::cs::controller::ActionCallback** and implement the **atst::cs::interfaces::IActionCallback** interface. The following methods are defined by this interface:

```
// report successful matching of the submitted Configuration
void done(pIAttributeTable data);

// report unsuccessful matching of the submitted Configuration
void abort(pIAttributeTable data);

// report on the current status of the submitted Configuration
// (does not terminate the action).
void report(pIAttributeTable data);
```

All of these methods are implemented by the **ActionCallback** base class and cannot be overridden. Controller subclasses can add additional operations to be performed by on successful completion of a submitted configuration by overriding the following methods from the **ActionCallbackAdapter** class:

```
// handle successful matching of the Configuration
void doDone(pIAttributeTable data);

// handle unsuccessful matching of the Configuration
void doAbort(pActionResponse response, pIAttributeTable data);

// handle a status report on the Configuration action
void doReport(const std:string& reason, pIAttributeTable data);
```

12.9 WRITING C++-BASED CONTROLLERS

12.9.1 The Controller Adapter Class and Source File

The class **atst::cs::controller::ControllerAdapter** subclasses **atst::cs::controller::Controller** and includes rudimentary implementations of the methods in the protected interface where you can attach application specific functionality to a controller subclass. While **ControllerAdapter** itself may be subclassed, a more practical use is to use its *source code* as a template for constructing your own controller subclass:

4. Copy **`$ATST/src/c++/atst/cs/controller/ControllerAdapter.h`** and **`$ATST/src/c++/atst/cs/controller/ControllerAdapter.h`** into the source directory for your new controller subclass, renaming it to your new controller name.

5. In that new source file, change:

    - the package name to your package,

    - the class name to your controller subclass name.

6. Now edit that source files to introduction the functionality required for your application.

12.9.2 Controller Example

The following is an example of a very simple controller. In short the controller implements a pseudo-state machine. The controller's health is related to the current state of the machine and only certain state changes are permitted. For the sake of brevity, the include block and the using block at the beginning applies to both the header file and the code file. The **ifndef**, and **define** bits have been left out. To see the actual source code look in an ATST Software Development Tree under $ATST/src/c++/cs/controller/ExampleController.[h|cpp].

```
#include "Controller.h"
#include "Runnable.h"
#include "ExampleController.h"
#include "Log.h"
#include "Event.h"
#include "ActionResponse.h"
#include "HealthStatus.h"
#include "Attribute.h"
#include "ErrorCode.h"
#include "Cache.h"
#include "Misc.h"
#include "Action.h"
```

```
#include <stdlib.h>
#include <time.h>
#include <try/memory>

using namespace atst::cs::interfaces;
using namespace atst::cs::data;
using namespace atst::cs::util;
using namespace atst::cs::services;
using std::tr1::shared_ptr;

//header file

class ExampleController;

class PostingThread : public atst::cs::util::threads::Runnable {
   public:
      PostingThread(ExampleController* parent);
      void doRun();
      void stop();
   private:
      ExampleController* controller;
      bool stopFlag;
};

class ExampleController : public Controller {
   public:
      int getState();
   protected:
      pActionResponse doSubmit(pIConfiguration configuration);
      bool doAbort(const std::string& configId);
      bool doCancel(const std::string& configId);
      bool doPause(const std::string& configId);
      bool doResume(const std::string& configId);
      void doInit() throw(LifeCycleChangeException);
      void doStartup() throw(LifeCycleChangeException);
      pActionResponse doAction(shared_ptr<IConfiguration> config);
      pActionResponse doCanRun(shared_ptr<IConfiguration> config);
      void doShutdown() throw(LifeCycleChangeException);
      //void doUninit() throw(LifeCycleChangeException);
      //void doRemove() throw(LifeCycleChangeException);
      pIAttributeTable doGet(pIAttributeTable table);
      void doSet(pIAttributeTable table);
      int calcDelay(pIConfiguration config);
   private:
      int state;
      shared_ptr<PostingThread> pt;
      static const int UNKNOWN_STATE;
      static const int AT_REST_STATE;
};

//source file

const int ExampleController::AT_REST_STATE = 0;
const int ExampleController::UNKNOWN_STATE = -1;

pActionResponse ExampleController::doSubmit(pIConfiguration config) {
   // if we get here we know that the technical architecture has
```

```
   // checked and the attributes are properly typed and within range
   // we just need to make sure that there is a new state attribute
   if (!config->contains("state"))
      return ActionResponse::missing("state");
   // and that the min time is less than max time
   int min;
   if (config->contains("minSimTime"))
      min = config->getInteger("minSimTime");
   else
      min = Cache::lookupWithDefault(
         Attribute::create("minSimTime", 500))->getInteger();

   int max;
   if (config->contains("maxSimTime"))
      max = config->getInteger("maxSimTime");
   else
      max = Cache::lookupWithDefault(
         Attribute::create("maxSimTime", 5000))->getInteger();

   if (min<=max) return Controller::ACTION_OK;
   // the params aren't bad they are just inconsistent
   else return ActionResposne::inconsistent("min must be less then max");;
}

pActionResponse ExampleController::doCanRun(pIConfiguration config) {
   int newState = config->getInteger("state");
   if (newState > state) return ACTION_OK;
   else return ActionResponse::reject(
         "New state is greater than current state.");
  // Default is assume no additional constraints on running
  return ACTION_OK;
}

pActionResponse ExampleController::doAction(pIConfiguration config) {
   shared_ptr<Action> a = getAction(config->getId());
   int newState = config->getInteger("state");
   int delay = calcDelay(config);
   int timeTake = 0;
   //wait for the action to complete
   while (timeTake < delay) {
      Misc::pause(50);
      timeTake += 50;
      if (a->paused()) a->pause(25);
      if (a->interrupted()) {
         if (a->wasAborted()) state = UNKNOWN_STATE;
         else if (a->wasCanceled()) state = AT_REST_STATE;
         else if (a->wasInterlocked()) state = UNKNOWN_STATE;
         else {
            state = UNKNOWN_STATE;
            Log::warn("Unrecognized interrupt reason!");
         }
         return a->interruptReason();
      }
   }
   state = newState;
   return ACTION_OK;
}
```

```
int ExampleController::calcDelay(shared_ptr<IConfiguration> config) {
   int min;
   if (config->contains("minSimTime"))
      min = config->getInteger("minSimTime");
   else
      min = Cache::lookupWithDefault(
         Attribute::create("minSimTime", 500))->getInteger();

   int max;
   if (config->contains("maxSimTime"))
      max = config->getInteger("maxSimTime");
   else
      max = Cache::lookupWithDefault(
         Attribute::create("maxSimTime", 5000))->getInteger();

   long delay = min;
   delay += rand() % (max - min);
   return delay;
}

bool ExampleController::doAbort(const std::string& configId) {
   // this controller can always abort
   // nothing to do here
   return true;
}

bool ExampleController::doCancel(const std::string& configId) {
   // this controller can always cancel
   // nothing to do here
   return true;
}

bool ExampleController::doPause(const std::string& configId) {
   // this controller can always pause
   // nothing to do here
  return true;
}

bool ExampleController::doResume(const std::string& configId) {
   // this controller can always resume
   // nothing to do here
   return true;
}

void ExampleController::doInit() throw(LifeCycleChangeException) {
   pt.reset(new PostingThread(this));
   srand(time(NULL));
}

void ExampleController::doStartup() throw(LifeCycleChangeException) {
   state = AT_REST_STATE;
   Misc::startDaemon(pt);
}

void ExampleController::doShutdown() throw(LifeCycleChangeException) {
   pt->stop();
```

```
}

//void ExampleController::doUninit() throw(LifeCycleChangeException) {}
//omitted because nothing needs to be done

//void ExampleController::doRemove() throw(LifeCycleChangeException) {}
//omitted because nothing needs to be done

void  ExampleController::doGet(pIAttributeTable table) {
   if (table->contains("state"))
      table->insert(Attribute::create("state", state));
}

void ExampleController::doSet(pIAttributeTable table) {
  // nothing to do until overridden
}

int ExampleController::getState() {
   return state;
}

PostingThread::PostingThread(ExampleController* parent) {
   controller = parent;
   stopFlag = false;
}

void PostingThread::stop() {
   stopFlag = true;
}

void PostingThread::doRun() {
   while (!stopFlag) {
      Misc::pause(500); //post at 2 hz
      shared_ptr<IAttributeTable> table = AttributeTable::create();
      table->insert(Attribute::create("state", controller->getState()));
      Event::post(controller->getName() + ".cstate", table);
   }
}

// The REGISTER macro is needed to allow containers to find and load
// this controller.  This should be included in a new file:
// $ATST/src/c++/cs/controller/ExampleControllerReg.cpp.

#include "cs/ClassLoader.h"
#include "cs/ExampleController.h"
#include "cs/Component.h"

REGISTER(
      atst::cs::controller::Controller,
      atst::cs::controller::ExampleController,
      "atst.cs.controller.ExampleController");
```

The makefile.gcc file in $ATST/src/c++/cs/controller/ will also need to be edited so that the new ExampleController builds correctly. Use $ATST/src/c++/atst/base/core/makefile.gcc as an example.

12.9.3 The Action Callback Adapter Class and Source File

The class **atst::cs::controller::ActionCallbackAdapter** subclasses **atst::cs::controller::ActionCallback** and includes rudimentary implementations of the **doDone**, **doAbort**, and **doReport** methods in the protected interface where you can attach application functionality. It serves the same roles to the **ActionCallback** class as **ControllerAdapter** serves to the Controller class.

12.9.4 Action Callback Adapter Methods

This section covers the roles of the protected methods that can be overridden in **ActionCallbackAdapter** subclasses to handle action responses that result from submitting configurations to other Controllers. Applications submit configurations to Controllers using one of the **submit** methods defined in the **IController** interface after connecting to the controller.

If the application *does not care* whether the resulting action completes successfully or not (e.g. the application is part of a processing pipeline), it may use the method **submit(IConfiguration config)**. Otherwise, one of the submit methods that allows attaching a callback should be used.

All action callbacks should subclass **atst::cs::controller::ActionCallback**, which provides the core behavior required of action callbacks by the technical architecture. A few methods are available for overriding to add functional behavior. The three key **ActionCallback** methods: **doDone**, **doAbort**, and **doReport** are called with the current configuration and the value of the event that led to the invocation of the callback (as an **AttributeTable**). In addition, **doAbort** also takes an **ActionResponse** object containing the reason for the aborting. The **doReport** method also takes a string containing the reason for the report.

***Method** doDone*

The **doDone** method is called internally by the **done** method when the target controller reports successful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding controller (or simple component, potentially!) in most cases some type of access to the controller will be needed (to set synchronization flags, for example). The methods **getController()** and **getComponent()**, described below, provide this access.

***Method** doAbort*

The **doAbort** method is called internally by the **abort** method when the target controller reports unsuccessful completion of the action. While it possible that the functional behavior needed here does not need access to information in the surrounding controller (or simple component, potentially!) in most cases some type of access to the controller will be needed (to set synchronization flags, for example). The method **getController()**, described below provides this access.

***Method** doReport*

The **doReport** method is called internally by the **report** method whenever a controller wants to issue a progress report on the processing of an action. Its use is not required.

***Method** getController*

The convenience method **atst::cs::interfaces::IController getController()** provides access to the controller that issued the command for the action being reported by this **ActionCallback**. The result will

likely need to be upcast to the appropriate controller subclass to call getters and setters to access information that needs to be shared between that controller and this callback. For example, the **doDone** method could be written as:

```
void MyCallback::doDone(pIAttributeTable newData) {
   dynamc_pinter_cast<MyController, IController>(
      getController)->signalDone(newData->getString("__.configId"));
}
```

The method signals successful completion of this action to the controller. Here, the **signalDone** method has been added as a public method to the controller subclass **MyController**.

***Method** getComponent*

The convenience method **atst::cs::interfaces::IComponent getComponent()** provides access to the Component that issued the command for the action being reported by this **ActionCallback**. The result will likely need to be upcast to the appropriate Component subclass to call getters and setters to access information that needs to be shared between that component and this callback. For example, the **doDone** method could be written as:

```
void MyCallback::doDone(pIAttributeTable newData) {
    dynamc_pinter_cast<MyComponent, IComponent>(
         getComponent()->signalDone(newData->getString("__.configId"));

}
```

The method signal successful completion of this action to the component. Here, the **signalDone** method has been added as a public method to the Controller subclass **MyComponent**.

This method ***must not be used*** if the submitter is a Controller subclass. Use **getController()** (above) instead.

***Method** getReason*

This convenience method provides access to the reason reported in an *abort* or *report* action response. A null is returned until an abort or report response has occurred.

***Method** getData*

This convenience method provides access to any application-specific data (if any) that has been associated with the latest action *report* response. It returns null until a report has occurred.

***Method** getConfig*

This convenience method provides access to the Configuration that was associated with the action.

***Method** isDone*

This convenience method returns true if the action callback has reported completion of the action (successful or unsuccessful).

***Method** isAborted*

This convenience method returns true if the action callback has reported failure of the action.

Method *waitForDone(long delay)*

This convenience method blocks until **isDone()** returns true.